

Exhibit D

Part 2 of 2

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

U·M·I

University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600

Order Number 9503201

**Simplified expression of message-driven programs and
quantification of their impact on performance**

Gürsoy, Attila, Ph.D.

University of Illinois at Urbana-Champaign, 1994

Copyright ©1994 by Gürsoy, Attila. All rights reserved.

U·M·I

300 N. Zeeb Rd.
Ann Arbor, MI 48106

BECKMAN00000191

**SIMPLIFIED EXPRESSION OF MESSAGE-DRIVEN PROGRAMS AND
QUANTIFICATION OF THEIR IMPACT ON PERFORMANCE**

BY

ATTILA GÜRSOY

**B.Sc., Middle East Technical University, 1986
M.S., Bilkent University, 1988**

THESIS

**Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1994**

Urbana, Illinois

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

THE GRADUATE COLLEGE

APRIL 1994

WE HEREBY RECOMMEND THAT THE THESIS BY

ATILA GÜRSOY

ENTITLED SIMPLIFIED EXPRESSION OF MESSAGE-DRIVEN PROGRAMS AND

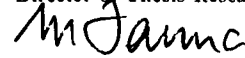
QUANTIFICATION OF THEIR IMPACT ON PERFORMANCE

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

THE DEGREE OF DOCTOR OF PHILOSOPHY

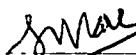


Director of Thesis Research

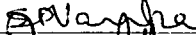
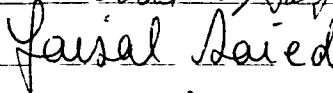
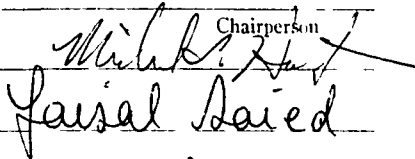


Head of Department

Committee on Final Examination†



Chairperson



† Required for doctor's degree but not for master's

©Copyright by
Attila Gürsoy
1994

Communication latency and unpredictable delays in remote response times constitute significant impediments to achieving high performance on massively parallel computers. Message-driven execution is a promising technique to improve the performance of parallel computations by overlapping these delays with useful computation. This thesis explores message-driven execution for improving performance of parallel programs. Programming in message-driven style is difficult due to the split-phase transactions it requires and due to the nondeterministic arrival of messages. We developed language constructs to express dependences between messages and computations in order to simplify expression of message-driven programs. Predicting the performance of message-driven programs via simulations is difficult because the arrival order of messages changes as the machine characteristics change. We developed a trace-driven simulation methodology based on the those language constructs. We also conducted an extensive performance study of message-driven programs.

ACKNOWLEDGEMENTS

I would first like to thank my advisor, Professor L.V. Kale, for his guidance and support during my doctoral research. He was a constant source of assistance, and his brilliant ideas were always stimulating. I also wish to thank Professor S.P. Vanka for being my co-advisor and for his support. I thank the other members of my committee: Professor M. Heath, D. Reed, and F. Saied, for their helpful recommendations.

Thanks to the members of the Parallel Programming Laboratory, including S.Krishnan, E.Kornkven, and A. Sinha, for providing a wonderful working environment and for listening to my practice presentations.

I also want to express my thanks to the Scientific and Technical Research Council of Turkey for providing me with the opportunity to pursue my doctoral studies at the University of Illinois.

Many of my friends deserve credit for their help. I would especially like to thank Maria Jose Gonzalez for her loving support and encouragement.

Finally, my profound thanks go to my parents: Ali and Ayşegül Gürsoy, and to my sister, Ayla Gürsoy. I cannot express with words how grateful I am for their constant support and love.

TABLE OF CONTENTS

Chapter

1	Introduction	1
1.1	Remote information access latency	2
1.2	Reducing the impact of latency: message-driven execution	2
1.3	Thesis objectives	4
1.3.1	Selection of computation domain for the performance study	5
1.4	Main contribution of the thesis	6
1.5	Thesis organization	8
2	Message-driven execution and Charm	10
2.1	Traditional SPMD model	11
2.1.1	Overlapping communication in SPMD	11
2.1.2	Traditional SPMD is inadequate to develop efficient large programs	14
2.2	Message-driven execution	15
2.2.1	Message-driven execution supports modularity	15
2.3	The potential benefits of message-driven execution	16
2.4	Emulating message-driven style in SPMD	20
2.4.1	Using nested if blocks	20
2.4.2	Using a global while-switch loop	21
2.5	Charm – a message-driven system	24
2.5.1	The Charm language	24
2.5.2	The Charm runtime	27
2.6	Related work on latency tolerance	28
3	Two obstacles	31
3.1	Programming difficulties of message-driven style	31
3.1.1	Nondeterministic message arrival	32
3.1.2	Obscure flow of control	35
3.2	Performance prediction: how to simulate message-driven computations	36
4	Controlling the complexity of message-driven programs	38
4.1	Divide-And-Conquer: a simpler context	39
4.1.1	Language definition	40
4.1.2	Data declarations	40
4.1.3	Blocks	41
4.1.4	Statements	42

4.1.5	An example	42
4.2	Dagger	44
4.2.1	Expecting a message	44
4.2.2	An example	45
4.2.3	Basic language	48
4.2.4	Dag-chare example	50
4.2.5	Extended language	50
4.2.6	Expressing loops in dag-chare	52
4.2.7	Reference numbers	58
4.2.8	Pipelining independent iterations of a loop	59
4.2.9	Multiple message entry points	60
4.2.10	Translation and runtime of Dagger	62
4.3	Message-driven libraries	66
4.3.1	Problems with libraries in SPMD style	66
4.3.2	Message-driven execution and library interface techniques	67
4.4	Related work	72
5	Simulating message-driven programs	79
5.1	Introduction	79
5.2	A sufficient condition for accurate simulation	81
5.3	Dagger programs and automatic trace generation	83
5.4	The parallel machine model	84
5.5	Simulator	88
5.5.1	Preprocessor	88
5.5.2	Parallel machine simulator	89
5.5.3	Interpreting the traces	90
6	Performance studies	92
6.1	Description of benchmarks	93
6.1.1	Synthetic benchmarks	94
6.1.2	Concurrent reductions	96
6.1.3	Harlow-Welch	101
6.1.4	Conjugate Gradient	110
6.2	Effects of network latency	114
6.2.1	Synthetic benchmarks	114
6.2.2	Concurrent reductions	117
6.2.3	Harlow-Welch	118
6.2.4	Conjugate Gradient	124
6.3	Effects of coprocessor	124
6.4	Random variations in latencies	132
6.5	Load balance versus critical path	133
6.5.1	Load balanced spanning trees and message-driven execution	139
6.5.2	Complementary spanning trees for multiple reductions	140
6.6	Effects of message scheduling	141
6.6.1	Preemptive scheduling	141
6.6.2	Priority-based scheduling	145

6.7 Conclusion	152
7 Conclusion	155
7.1 Future work	157
Bibliography	159
Vita	168

LIST OF TABLES

5.1	Machine parameters	86
5.2	Events in a when-block trace	89
6.1	CG results on NCUBE/2.	113
6.2	Communication parameters for the coprocessor experiment.	127
6.3	Communication parameter settings in the variable latency test.	132
6.4	Effects of branching factor.	140
6.5	Effect of complementary spanning trees.	141

LIST OF FIGURES

1.1	Remote information access latency.	3
2.1	Simple SPMD codes (a) with message passing primitives (b) with library/module calls.	12
2.2	Rearranging send and receives (a) a sample code (b) rearranged code.	13
2.3	Processor utilization for (a) naive code (b) rearranged code.	13
2.4	SPMD modules cannot share the processor time.	14
2.5	Message-driven modules share the processor time.	16
2.6	s independent threads.	17
2.7	Overlapping latency.	17
2.8	Latency tolerance.	19
2.9	Effect of overhead.	20
2.10	Using nested if's to simulate message-driven execution.	22
2.11	A global while-switch construct to simulate message-driven execution.	22
2.12	Chare definition.	25
2.13	A branch office – ring program.	26
2.14	Multiple modules in Charm (a) module M2 accesses entities in module M1 (b) interface module for M1 M1.interface.	28
3.1	Incorrect message-driven code.	33
3.2	Correct message-driven code.	34
3.3	Flow of control (a) SPMD (b) pure message-driven (c) partial order.	36
4.1	Divide-and-conquer node definition.	41
4.2	Node definition to compute Fibonacci numbers.	43
4.3	A message triggers a computation (a) in pure message-driven (b) in Dagger.	44
4.4	Matrix multiplication chare.	46
4.5	Matrix multiplication dag-chare.	47
4.6	Dag-chare template.	48
4.7	Dag-chare illustrating adaptive overlapping.	51
4.8	Red-black Gauss-Seidel (a) partitions (b) dependences on one processor.	53
4.9	Dag-chare for Gauss-Seidel red-black relaxation.	55
4.10	Jacobi (a) partitioning (b) dependences on one processor.	56
4.11	Partial dag-chare for Jacobi relaxation.	56
4.12	Out of order messages.	57
4.13	Loop structure in (a) red-black (b) Jacobi.	58
4.14	Correct Jacobi relaxation with reference numbers.	59

4.15	Pipelining loop iterations.	60
4.16	A reduction dag-chare illustrating multiple message entries.	62
4.17	Translation of a dag-chare.	63
4.18	Library call.	68
4.19	Using the reduction library.	71
4.20	Distributed Processes example.	74
4.21	Ada select statement.	76
5.1	A dag computation	82
5.2	The parallel machine	85
5.3	Sending a message	85
5.4	Message queues	87
5.5	Simulation system	88
6.1	Synthetic benchmark Wave (a) message-driven (b) SPMD.	95
6.2	Synthetic benchmark Mlib (a) message-driven (b) SPMD.	97
6.3	Concurrent reductions (a) SPMD (b) message-driven.	98
6.4	Pipelining and effects of overhead.	100
6.5	Concurrent reductions: effect of number of partitions on NCUBE/2.	101
6.6	Tolerating latency: concurrent reductions on NCUBE/2.	102
6.7	One time step of Harlow-Welch algorithm.	102
6.8	Decomposition of the computational domain.	102
6.9	Jacobi message-driven code.	104
6.10	Red-Black1 message-driven code.	105
6.11	Dependences of the Stone's method.	107
6.12	Stone's method: effect of pipelining.	109
6.13	Stone's method on NCUBE/2 (a) without reductions (b) with reductions.	111
6.14	Stone's method message-driven code.	112
6.15	Dependences in the modified CG method.	113
6.16	Effects of network latency: Synthetic Wave.	115
6.17	Effects of network latency: Synthetic Mlib.	116
6.18	Effects of network latency: Synthetic Mlib with varying computation load.	117
6.19	Concurrent reductions: effect of network latency α_{net} , and β_{net}	119
6.20	Harlow Welch with Jacobi: effect of network latency α_{net} , and β_{net} (a) $\beta_{net} = 0$ (b) $\beta_{net} = 1$ (c) $\beta_{net} = 10$	120
6.21	Harlow Welch with Red-Black1: effect of network latency α_{net} , and β_{net} (a) $\beta_{net} = 0$ (b) $\beta_{net} = 1$ (c) $\beta_{net} = 10$	122
6.22	Harlow Welch with Red-Black2: effect of network latency α_{net} , and β_{net} (a) $\beta_{net} = 0$ (b) $\beta_{net} = 1$ (c) $\beta_{net} = 10$	123
6.23	Harlow Welch Stone: effect of network latency α_{net} , and β_{net} (a) $\beta_{net} = 0$ (b) $\beta_{net} = 1$ (c) $\beta_{net} = 10$	125
6.24	Conjugate Gradient (model):effect of network latency α_{net} ,and β_{net} (a) $\beta_{net} = 0$ (b) $\beta_{net} = 1$ (c) $\beta_{net} = 10$	126
6.25	Effect of the coprocessor - concurrent reductions. Sum of processor and copro- cessor delays (a) 1000 (b) 10000 units.	129
6.26	Effect of the coprocessor - multiplane Jacobi. Sum of processor and coprocessor delays (a) 1000 (b) 10000 units.	130

6.27 Effect of the coprocessor - multiplane Stone's method. Sum of processor and coprocessor delays (a) 1000 (b) 10000 units.	131
6.28 Variable network latencies - Synthetic Wave.	133
6.29 Variable network latencies - Synthetic Mlib.	134
6.30 Variable network latencies - Concurrent reductions	135
6.31 Variable network latencies - Multiplane Jacobi.	136
6.32 Variable network latencies - Multiplane Stone.	137
6.33 Variable network latencies - One plane Red-Black2.	138
6.34 Load balance of spanning trees.	140
6.35 A scheduling problem.	143
6.36 Reductions with interrupts.	144
6.37 Reductions with higher priority.	146
6.38 Two different schedules.	148
6.39 Multi-plane Jacobi (a) load index (b) concurrency index.	149
6.40 Multi-plane Jacobi : FIFO versus priority.	149
6.41 Multi-plane Stone (a) load index (b) concurrency index.	150
6.42 Multi-plane Stone : FIFO versus priority.	150
6.43 Greedy scheduling example (a) FIFO (b) forced static scheduling.	151

Chapter 1

Introduction

Parallel computers potentially offer much more computational power than uniprocessor machines. Although recent advances in VLSI and microprocessor technology have improved the performance of uniprocessor systems, there are many problems that require computational power well beyond what uniprocessor systems can provide. Therefore, substantial effort is devoted to developing efficient and cost effective parallel machines.

Currently, there are many commercial parallel computers with very high peak performance. Machines with tens of gigaflops capacity are already available, and teraflops machines are expected in the next decade. However, their performance falls substantially short of peak on many real life applications. A number of factors contribute to this performance loss. For example, the sequential performance of individual processors itself may fall short of the peak power because of cache performance and inability to exploit fully features such as multiple instruction issues, pipelining, etc. Hopefully, this problem can be solved eventually by better compilers, use of optimized libraries such as BLAS routines [28], and occasional use of assembly code. In any case, this factor is not specific to parallel computing, as performance of workstations is also affected by it.

An important factor that is specific to parallel computing is the loss of performance due to communication latencies and processor idling due to load imbalances and critical paths. These factors and methods for overcoming them are the focus of this thesis.

1.1 Remote information access latency

A parallel computation is a collection of co-operating processes. The processes and data are distributed across the parallel machine. Typically, these concurrent processes interact with each other during the course of computation. This interaction might be due to synchronization or to data access. In either case, a process needs information that is created or stored by some other processes. If the needed information is on some other processor, then access to this remote information will be slower than that to local information. This slowdown generally occurs for two reasons.

One of the reasons is the delay introduced by the communication network. The communication networks of large private memory machines are usually multi-hop switching networks. Messages experience delay of transmission, routing, and buffering. This delay, which we will call *communication delay*, is defined as the time interval between the moment data enter into the communication network and the time the data become available at the destination.

The other source that contributes to the remote information access latency is the delay in the creation of the information at the remote site (Figure 1.1). Process 1 sends a request for some particular information. When process 2 receives the request, it performs the required service, creates the information and sends it back. The time interval between the arrival of the request and the completion of the requested service is the *response delay*. The *response delay* can be longer than the service time itself and is often unpredictable. For example, the processor might be scheduled for some other task, or cache performance might delay processing. Response delays take place not only in request-and-respond type of interaction; it equally affects prearranged communication (i.e., prepare-and-send) where no request is necessary.

These two types of delays together form the remote information access latency, simply referred to as *latency*. Minimizing the impact of this latency is a major objective in parallel programming on parallel machines, and particularly on massively parallel scalable machines.

1.2 Reducing the impact of latency: message-driven execution

The impact of latency can be reduced in several ways. On the hardware side, this is addressed by designing architectures that attempt to reduce the communication latency to the minimum. The ALLCACHE architecture of the KSR-1 machine, and the message-processor architecture

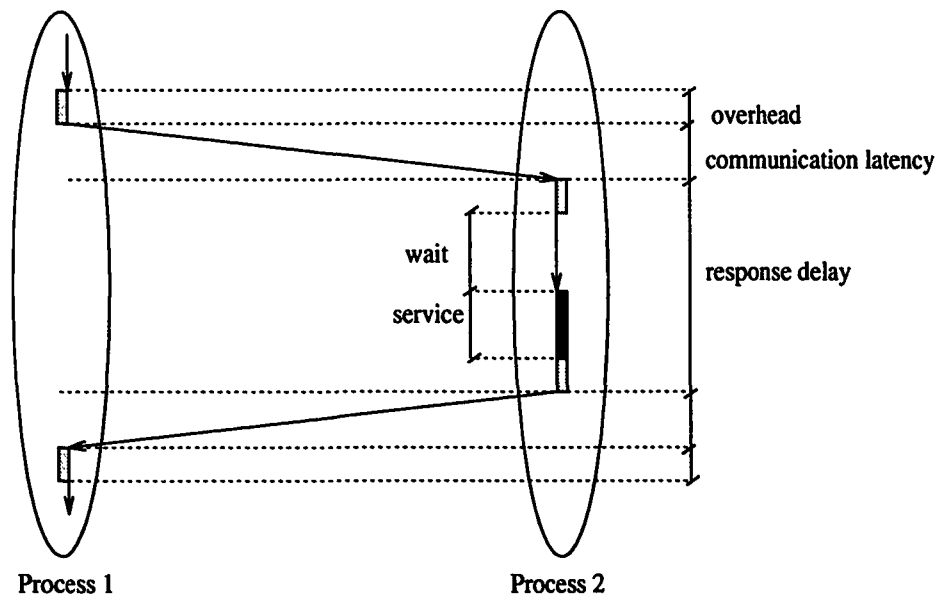


Figure 1.1: Remote information access latency.

of J-Machine [22] are examples of these attempts as well as the continuous evolution of communication hardware in the traditional architectures of Intel and NCUBE machines [77]. However, physical reality dictates that remote access will always be significantly slower than local access (overhead can be reduced by better operating system support such as Active messages [31], SUNMOS).

Since latency cannot be eliminated completely, one may try to minimize the number of remote data accesses. This requires the placement of computations and data to be reorganized so that data accesses are localized as much as possible [43, 92].

A complementary approach to reduce the impact of latency, on the software side, is to overlap the delay with some useful computation. This idea — doing something else useful while waiting for data — has been used at different levels in computer systems. Multiple functional units of a cpu may overlap independent operations. For example, while a floating-point computation unit is busy, the next instruction and data could be fetched from the memory. In the prevalent parallel programming paradigm (the traditional SPMD paradigm, see Section 2.1), such overlap is enhanced by moving `sends` earlier and moving `receives` later. However, this

strategy is not sufficiently adaptive to the eventualities that may arise at runtime (see Chapter 2).

Message-driven execution is a promising technique in this regard. In the message-driven execution style (which is distinct from message-passing), there are typically many processes per processor. A process does not block the processor on which it is running while trying to receive a message. Instead, the system activates a process when there is a message for it. This improves latency tolerance in two ways. First, when one process is waiting for data from a remote process, another ready process may be scheduled for execution. Second, even a single process may wait for multiple data items simultaneously, and continue execution whenever any of the expected items arrive. If there are multiple subcomputations, each with t_1 and t_2 running time respectively, the message-driven running time is $t_1 + t_2 - t_{\text{overlap}}$ as explained in Chapter 2. As we will see later, this leads to adaptive overlap and modularity.

1.3 Thesis objectives

This thesis will explore the message-driven execution technique for tolerating communication latencies. The major objectives of this thesis are:

Simplification of expression of message-driven execution:

Although message-driven execution imparts the benefits alluded to above, it often extracts a price in the form of apparent program complexity. The *split-phase* or *continuation-passing* style of programming that it requires is sometimes non-intuitive and obfuscates the flow of control. As the system may execute messages in the order it receives them (as opposed to a deterministic order imposed by sequential receive statements), the programmer must deal with all possible orderings of messages. This involves complex reasoning about which message-orderings will not arise, which are harmless, and which must be dealt with by buffering, counters, and flags. Therefore, it is desirable and important to simplify the expression of message-driven programs, possibly via new languages or annotations.

Performance prediction of message-driven execution:

Predicting the performance of message-driven computations via simulations is necessary to evaluate the benefits of message driven execution under various machine characteristics. In such computations, the sequence in which messages are processed is not fully specified.

As a result, at runtime messages may be executed in different sequences depending on runtime conditions. This makes simulation and performance prediction of message-driven programs a challenging task.

Evaluation of the performance impact of message driven execution:

The ability to overlap computation with communication provides message-driven execution with the potential to improve the performance of a parallel program. However, this potential benefit needs to be established by performance studies. Such performance studies will help to determine:

- the effectiveness of message-driven execution in increasing performance,
- the factors that influence the performance of message driven programs,
- the conditions under which these performance benefits are realizable.

1.3.1 Selection of computation domain for the performance study

The performance studies in this thesis involve simulation and real machine implementation of selected computations. For dynamic tree-structured computations such as heuristic search, divide-and-conquer, game tree search, and evaluation of functional and logic languages, the performance advantages of message-driven execution is intuitively quite clear. In such problems, the message-driven strategy can adapt to the unpredictable nature of communication latencies and computational variations better than the conventional message passing style. For example, while evaluating a node of an and-or tree, one cannot predict if the node will be a simple leaf node or will lead to a large subtree. The structure of the tree is data dependent and dynamic. Due to the unpredictable nature of parallelism, one must assign many tree nodes per processor. As each node has a complex behavior, it is natural to treat each node as an individual object. The system must then process every incoming message, decide whether it involves creation of a new object (*i.e.*, creation of a node), or it is a response from one of the subproblems of an and-or tree node, and activate the corresponding node to process this message. Each node itself must be willing to accept solutions (or a failure to find solutions) for any one of the multiple subproblems it may have started and create new subproblems that may depend on such solutions. All of this naturally leads to message-driven style for implementation. For

these reasons, many researchers in these areas have used message driven-execution, implicitly or explicitly [19, 63, 82].

A more interesting question, therefore, is whether message-driven execution is beneficial for static computations — computations whose overall structure is known ahead of time. Therefore, investigating the performance advantages of message-driven execution for such computations will help answer an interesting and important question; if such advantages are established, it will lead to a stronger endorsement of the message-driven execution. Despite their static nature, communication latencies and their associated unpredictability make these computations potential candidates for message-driven execution.

Computational Fluid Dynamics (CFD) appears to be a proper area to conduct the performance studies because (a) many CFD algorithms have a static structure, and (b) it is an important application area for parallel computing. A review of many CFD applications reveals that many of them use a few numerical kernel algorithms that have regular computational and communication structure [4, 7, 10, 11, 24, 42, 75]. We chose a few of these core problems for our study as described in Chapter 6.

1.4 Main contribution of the thesis

- The thesis develops a set of arguments and lines of reasoning with examples highlighting the differences between SPMD and message-driven programs and argues for the potential advantages of message-driven execution.
- The thesis research involves development and implementation of a language, Dagger, for simplified expression of message-driven programs.
- A methodology is devised for trace-driven simulation of message-driven programs, and a simulator based upon it is implemented.
- The thesis includes an extensive performance study involving several synthetic and real benchmarks, and performance measurement on real machines as well as simulations with varying machine parameters.

The idea of message-driven execution was first proposed by Hewitt [53] and later was elaborated by Agha [1]. The work on dataflow also relates to the same idea at a finer grain and

at the hardware level. However, there has not been an extensive comparison of this approach with the now dominant message passing model (receive based SPMD model that is elaborated in Chapter 2). We argue that modularity and efficiency – in the form of overlapping communication with computation – can be achieved much more easily in message-driven execution than in the traditional message passing paradigm. We also show why it is not adequate to emulate message-driven execution with the message passing style. Section 2.6 discusses other approaches related to message-driven execution with the Charm system. Charm was one of the first message-driven systems and is used as an implementation substrate in this thesis.

Programming in message-driven style is difficult due to the split-phase style of programming it requires and due to the nondeterministic arrival of messages. Therefore, the synchronization of messages and subcomputations in a process is necessary to maintain the integrity of the computation. Dagger expresses the partial orders among subcomputations and messages within an object, and yet retains the advantages of message-driven execution (adaptive scheduling of subcomputations based on message arrivals).

The issue of synchronization has been dealt with extensively in past research starting from the mid 1970's. However, many of the schemes require the caller of a method to block until the called object finished its service. In most of these approaches, the server is also blocked in a variety of contexts. None of these features can be used effectively to express adaptive overlap of different computations based on availability of data, which is an important requirement for message-driven execution. The distributed process model by Hansen [50] comes closest to Dagger in many respects. This model as well as others are compared to Dagger in Section 4.4. More recent work on the Actor model and other concurrent object oriented languages also address the issue of local synchronization within an object. These studies were also summarized in Section 4.4.

Although trace-driven simulation is a very efficient method for predicting performance of parallel programs under varying runtime conditions, such simulation methodology has not been applied to message-driven programs or to programs that use wild-card receives. The basic difficulty in such simulations is the fact that if two messages are received in different orders under different runtime conditions, then the behavior of the program may be altered in a way that cannot be reconstructed based on the traces obtained with the first sequence. The thesis presents a methodology for simulation of message-driven programs that exploits the

dependence information provided by Dagger at compile time combined with a special method for obtaining runtime traces. We know of no other results that can carry out performance analysis or simulations in a trace driven manner for such programs. Further, existing systems do not handle the case of a different message ordering during simulation.

The thesis includes an extensive performance study involving several synthetic and real benchmarks, and performance measurement on real machines as well as simulations with varying machine parameters. This study establishes the performance advantages of message-driven execution clearly and quantifies its benefits and cases in which these are realized. It also leads to new criteria for algorithm design for message-driven programs and emphasizes the importance of message scheduling.

1.5 Thesis organization

The thesis is organized as follows. Chapter 2 discusses and contrasts message passing and message-driven paradigms. It establishes preliminary evidence for the utility of message-driven execution. The Charm language, a message-driven system that we have chosen as a medium for this study, is presented. Chapter 3 elaborates difficulties encountered using the message driven style. These difficulties center around programming complexity and performance prediction via simulations. Chapter 4 describes the techniques we have developed as a part of this thesis for simplifying message-driven programming. These involve design and implementation of coordination languages to control the programming complexity. A coordination language to support message-driven computations in simpler contexts such as functional (divide-and-conquer) computations is described, as well as the Dagger language which allows easier expression of general purpose message-driven programs. Chapter 4 continues with the discussion of message-driven libraries. Libraries, in general, ease program development. Issues involved in the interface between libraries and programs in a message-driven environment are discussed. Dagger also provides a basis to carry out trace-driven simulations of such programs, eliminating the difficulty discussed in Chapter 3. Chapter 5 describes a trace-driven simulation methodology for message driven computations. The information necessary to carry out such simulations is identified, and a method for extracting such information from Dagger programs during compile-time and program execution is described. A general and parameterized model of parallel machines embodied

in the simulator is discussed. In Chapter 6, we evaluate the performance of message-driven computations on the selected problems. The evaluation is carried out with simulations and real machine implementations. The impact of latency on message-driven and message-passing implementations of the selected problems is observed, and the effect of various runtime conditions and parameters on their behavior is studied. The chapter concludes with message-driven algorithm design techniques that help improve the performance of message-driven algorithms. Chapter 7 presents conclusions.

Chapter 2

Message-driven execution and Charm

This thesis is concerned mainly with distributed-memory machines and with explicitly parallel programming. Most large parallel machines currently available are distributed-memory machines, such as nCUBE/2, CM-5, Intel/Paragon (although there are also large scalable shared-memory machines, such as KSR/1).

There are many programming models for programming parallel machines. A data parallel programming model with languages such as Fortran D and HPF [35] is useful for array-oriented applications. There are also parallel languages such as functional and logical languages that can be used to program these machines. However, neither of these models is general enough to apply easily to a broad class of applications, and therefore explicitly parallel languages are widely used. In explicitly parallel languages, the programmer explicitly specifies the partitioning of the application into parallel parts and often the mapping of the parts to processors. Such parallel languages include Linda [14], with a specialized or idiosyncratic programming model. However, the predominant paradigm used for programming parallel machines is provided by the *traditional SPMD* model, which is supported by vendors of parallel machines in their operating systems. Even data parallel and functional languages are often implemented using the traditional SPMD model as their backend.

This chapter will examine the traditional SPMD model and show its inadequacies in dealing with communication latency and adaptive overlap. It will elaborate on the message-driven

execution model and show how it can handle latency better, and will motivate, using analysis of a simplified computation model, the potential benefits of message-driven execution. It is possible to emulate message-driven execution within the extended SPMD style programming using asynchronous receives. How this can be done, and why this is not an adequate solution is shown in Section 2.4. Instead, a language that incorporates message-driven execution in its model is better suited for this purpose. Charm, a message-driven language, is used as a substrate for the work in the rest of the thesis. Other message-driven languages, as well as other related techniques for tolerating communication latencies, are described in Section 2.6.

2.1 Traditional SPMD model

The SPMD — single program multiple data — model simplifies program development by using a simple model for internal synchronization and scheduling. The phrase SPMD has been used with somewhat different meanings by different authors [40, 43, 69, 80, 84]. In the SPMD model, as used in this thesis, there is one process per processor (usually all processes are executing the same program). Communication among processes (hence processors) is usually with blocking primitives. Messages have tags, and the `receive` primitive blocks the processor until a message with a specified tag arrives (of course, there is no reason not to use non-blocking communication occasionally if it does not complicate the code). Moreover, we use “traditional SPMD model” to mean strict usage of blocking receives.

A single thread of control and blocking receives makes the programming of these machines relatively easy. Most of the synchronization requirements of the computation are handled implicitly by the blocking receives. For example, consider the code in Figure 2.1-(a). The computation of `t1` and `t2` needs the remote information `a` and `b`. Since the `recv` statements will not allow the processor to proceed until the required messages are received and made available to the code, the correct computation of `t1`, `t2` and `t3` is guaranteed.

2.1.1 Overlapping communication in SPMD

The simplicity of the flow-of-control attained in SPMD is at the expense of idling processors. After issuing a blocking receive, the processor must wait idly for the specified message to arrive. This wait may not always be dictated by the algorithm, *i.e.*, the algorithm may have more

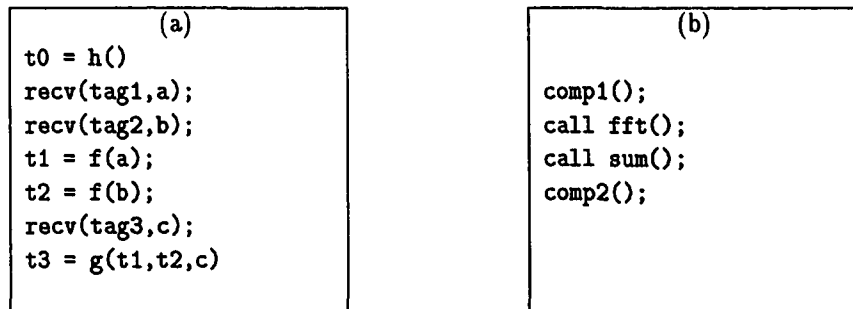


Figure 2.1: Simple SPMD codes (a) with message passing primitives (b) with library/module calls.

relaxed synchronization requirements. Yet the usage of blocking primitives forces unnecessary synchronization and may cause idle time.

This idle time can be decreased by rearranging the send and receive operations. This involves moving the sends ahead and postponing the receives as much as possible in the code, as illustrated by the following example. A sample SPMD code is shown in Figure 2.2-(a). The code has a pre-communication phase (t_1, t_2) and a post-communication phase (t_4, t_5). Between these computation phases, it communicates with some remote processors by sending and receiving messages. This type of structure is common in SPMD programs that are either written by a programmer or automatically generated by a compiler [40]. The utilization of a processor during the execution of this code is shown in Figure 2.3-(a). The code is running on processor P1. After completing t_1 and t_2 , the send is initiated. Then the processor starts the receive operation, which blocks the processor. Therefore, processor P1 is forced to stay idle during the communication latency of both send and receive operations (Although the algorithm does not require the first call to `g` to wait for the `recv` to finish, the blocking semantic of `recv` forces this synchronization).

Figure 2.2-(b) shows the rearranged code that overlaps the communications latency with some useful computation. The data dependencies among the computations reveal that the send can be issued just after the computation of t_1 , and t_3 is needed only for the computation of t_5 . Figure 2.3-(b) shows the processor utilization after moving the send and receives appropriately. Processor P1 initiates sending the message and continues with the computation of t_2 and t_4 while the message transfer continues in the communication network. The remote processor

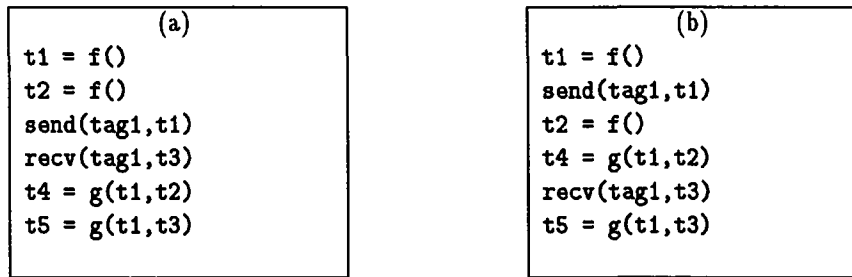


Figure 2.2: Rearranging send and receives (a) a sample code (b) rearranged code.

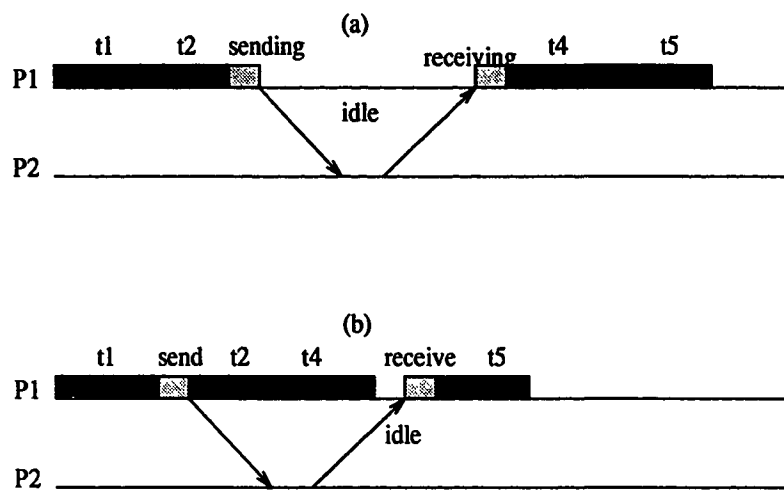


Figure 2.3: Processor utilization for (a) naive code (b) rearranged code.

P2 receives the message (request) and responds with the result message. The latency of this message is overlapped by the computation t4 on processor P1 since it invokes the receive operation after t4.

For this simple example, the local rearrangement of communication achieves the desired objective – increasing the utilization of processors. However, this strategy cannot handle cases with more complex dependencies and unpredictable latencies. For example, consider again the code in Figure 2.1-(a). The code contains two receives followed by computations t1, t2, and t3. t1 and t2 are independent of each other and need data from different receives. One of the receives can be postponed to overlap latency of the other one. If the latencies for these receives

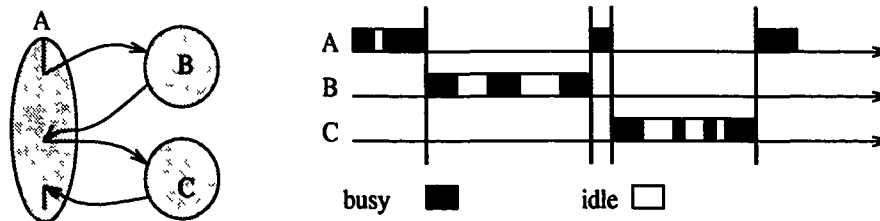


Figure 2.4: SPMD modules cannot share the processor time.

are known *a priori*, then the one with longer latency may be postponed. However, in general, unpredictable communication delays make it difficult to know the latencies in advance. In case of such unpredictability, if the one that is postponed turns out to be the wrong one at runtime, then the idle time on the processor would be the same as the time for the original code. Thus, the SPMD model cannot adapt to such runtime conditions by rearranging the send and receive operations.

2.1.2 Traditional SPMD is inadequate to develop efficient large programs

Although SPMD model can achieve limited performance improvements as discussed in the previous section, it cannot overlap computation and communication across modules and libraries. A module or library is defined as an independently developed program that can be called from other programs, as in Figure 2.1-(b), which invokes two modules (`fft` and `sum`). In the SPMD style, invocation of another module passes the flow of control to that module. Until that module returns control, the calling program cannot do anything. Therefore, the idle times that a module experiences cannot be overlapped with computation from another module¹. In Figure 2.4, for example, module A invokes two other modules B and C. Module A cannot activate B and C concurrently even if the computations in B and C are independent of each other. As a result, the processor time is not fully utilized, as illustrated in the same figure.

Notice that this problem is independent of communication latency. Even if communication latency were to be zero, each of the modules may have idle times on individual processors due to critical paths and load imbalances in them.

¹See Section 2.4 for a discussion of how one can use non-blocking message passing primitives for this purpose in SPMD style and why this is not adequate

Despite its simplicity, the traditional SPMD model is far from being a programming model for developing efficient large parallel applications for these reasons. The message-driven execution helps to solve these problems.

2.2 Message-driven execution

Message-driven execution, in contrast to the SPMD model, supports many small processes (or objects) per processor. These processes are activated by the availability of messages that are directed to them. At this level of description, it suffices to say that each process has a state and a set of functions (methods) for dealing with incoming messages. When a message arrives for a particular process, the system eventually activates the process. Then the process, depending on the content and type of the message, executes the appropriate method.

The system has a pool of incoming messages, because during the execution of a previous process many messages may have arrived. After a process suspends itself, the system selects one of the messages from the pool to be processed next. This selection procedure – message scheduling – can be a simple strategy such as FIFO, or can be a more complex strategy such as prioritized scheduling.

2.2.1 Message-driven execution supports modularity

Message-driven execution overcomes the two difficulties experienced by the SPMD model. It can effectively overlap latency with useful computation adaptively:

- within a module
- across the modules

The rearrangement of send and receives in SPMD, for example, failed to achieve the overlap adaptively (the example in Figure 2.2). Message-driven execution could compute either t_1 or t_2 in Figure 2.1-(a) depending on which message arrived first (or whichever message is made available for processing by the message-scheduler), hence, adapting itself to the runtime conditions. The computation of t_3 , however, now requires internal synchronization necessitated by message-driven style (*i.e.*, it can be computed only after t_1 and t_2 are both calculated). We will discuss a method for expressing such synchronization in the next chapter.

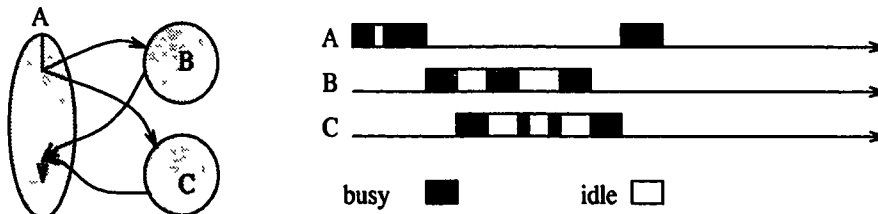


Figure 2.5: Message-driven modules share the processor time.

The message-driven paradigm, in addition, allows different modules that might have some concurrent computations to share processor time. Consider the computation discussed in Section 2.1.2. Assuming that modules B and C do not depend on each other, the idle times on a processor can be utilized by another module if it has some work to do. Such a scenario is illustrated in Figure 2.5. Module C gets processor time (by virtue of having its message selected by the scheduler) while B waits for some data, and vice versa, thus achieving a better overlap than the SPMD program in Figure 2.4.

2.3 The potential benefits of message-driven execution

A performance analysis of a simple application model of message-driven execution is now developed to motivate the performance benefits of message-driven execution.

In this model, each processor performs k computations. Each computation is of length t_p time units and issues one remote data access at the end (sending and receiving a message). Each remote access has a latency of l time units. Some of these computations can be executed concurrently, as they do not depend on each other. The number of such concurrent computations is denoted by s , as illustrated in Figure 2.6. In practice, these s concurrent computations may arise at a particular time due to independent computations within a module or computations from different modules.

In the SPMD model, these computations will be executed sequentially because, as explained previously, the SPMD model is not adaptive to variations in latencies and blocks for receives, and it does not allow computations from different modules to be interleaved. Therefore, it is reasonable to assume that after every computation of length t_p , in the SPMD style, the processor blocks for l time units. The completion time of the whole computation for SPMD

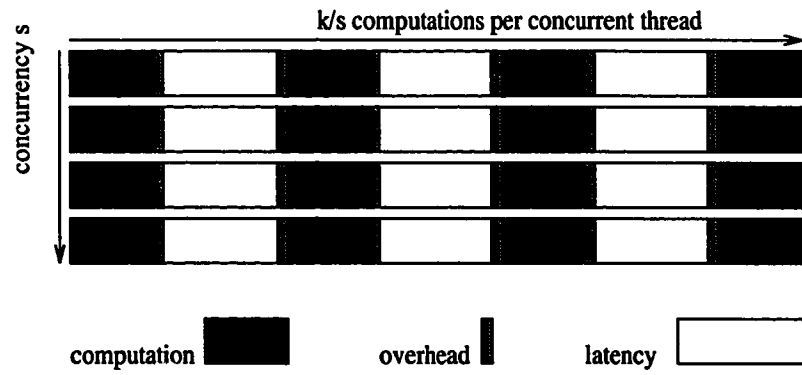


Figure 2.6: s independent threads.

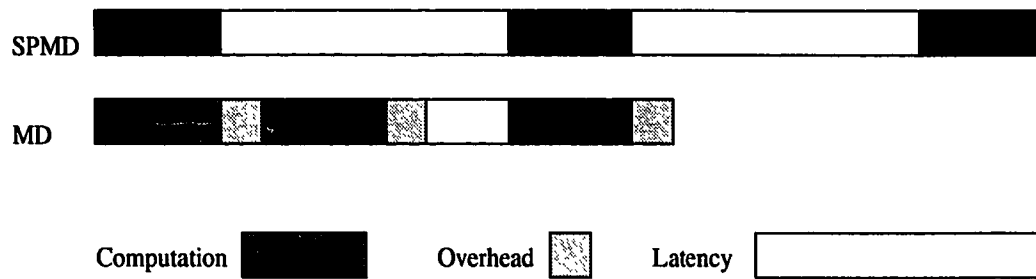


Figure 2.7: Overlapping latency.

can be written as:

$$T_{SPMD} = k(t_p + l)$$

Message-driven execution, on the other hand, is capable of activating any computation that is ready to be executed. In order to model message-driven execution, one more parameter, t_o , is introduced to reflect the additional overhead of message-driven execution. This overhead is due to context switching and message buffering required during the execution of the program.

The completion time for the message-driven case depends on how much of the latency is overlapped. If all of the latency is overlapped with useful computation, then the completion time would be $k(t_p + t_o)$. This happens if the latency is less than the time required to complete s concurrent computations. If the latency is larger than that, then the time to complete s concurrent computations will be $t_p + l$, where t_p is the time required for the first computation out of s computations. The time for the other $s - 1$ computations will be masked by this latency. Therefore, the completion time for the message-driven case is:

$$T_{md} = \begin{cases} k(t_p + t_o) & \text{if } s > \frac{l+t_p}{t_p+t_o} \\ \frac{k}{s}(t_p + l) & \text{otherwise} \end{cases}$$

In order to determine the effects of the latency, the completion time of the model is plotted versus latency in Figure 2.8 for both SPMD and message-driven style. The overhead t_o for the message-driven execution has been kept zero. The latency is in terms of the computation time. The plot shows the latency tolerance curves for different concurrencies, s . There are a number of important observations here. The completion time of the message-driven execution remains constant as the latency increases up to a certain point, while the time for the SPMD version increases linearly with the latency. In the flat phase of the message-driven version (i.e., the region left of the knee), message-driven execution completely overlaps the latency with computations. When the latency becomes too large, then the performance of the message-driven execution also degrades. However, it is asymptotically still better than the SPMD in the sense that the slope of the curve is lower, because it continues to overlap some parts of the latency.

In the second graph, Figure 2.9, the effects of the overhead are plotted. In this case, s is kept fixed and completion time curves for different values of t_o are plotted against the latency.

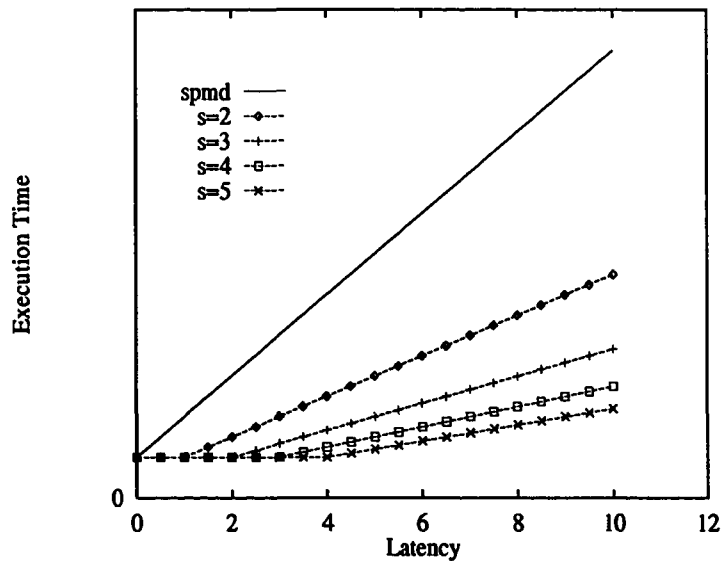


Figure 2.8: Latency tolerance.

The effect of overhead is an upward shift in the first phase (the phases where the latency is small enough to overlap completely). However, for larger latencies, the overhead is absorbed in the latency time also.

This simple model shows that it is worthwhile exploring message-driven execution for developing parallel algorithms. However, there are numerous factors affecting performance that this model has ignored, such as the scheduling policy, irregular computation times and latencies, and unbalanced load. To predict the performance of message-driven computations under these conditions becomes very difficult, if not impossible, with analytical methods. What is needed is an empirical study of the potential benefits of message-driven execution over a range of architectural parameters and application programs. Such a study is one of the foci of this thesis. To this end, we have developed a simulation framework to study the performance of message-driven computations more accurately (see Chapter 5).

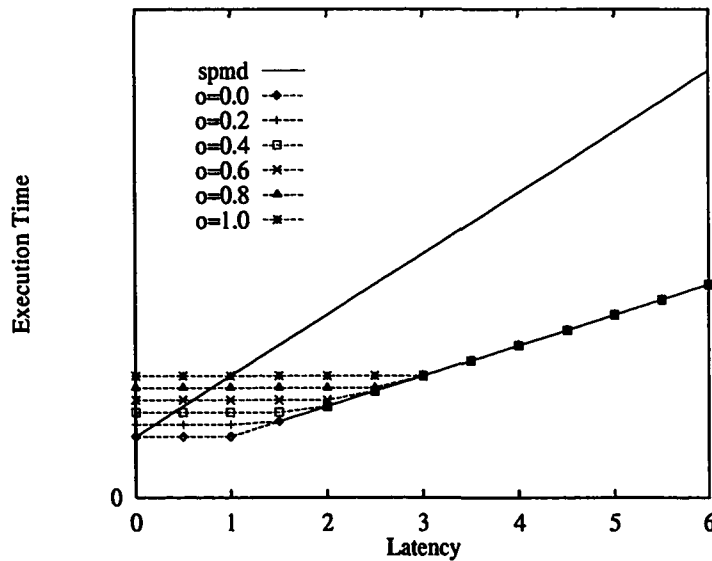


Figure 2.9: Effect of overhead.

2.4 Emulating message-driven style in SPMD

We have defined message-driven execution and shown its potential performance benefits. In this section, we will discuss how message-driven execution can be emulated within the SPMD context and examine the adequacy of such an emulation.

The emulation of message-driven execution involves usage of nonblocking message passing primitives, particularly the nonblocking receive primitive. The examples presented in this section will use a primitive operation called *probe* and the blocking receive primitive instead of nonblocking receive. The *probe(tag1)* function checks if a message with the tag *tag1* has arrived. A probe followed by a blocking receive (if probe succeeds) is equivalent to nonblocking receive for our purposes.

2.4.1 Using nested if blocks

A simple approach to incorporate message-driven ideas in SPMD programs is to use nested if statements and nonblocking receives to replace some of the “blocking receive and compute” sequences. These sequences are the part of the code where the rearrangement of receives does not help for dynamic situations – as in the example of Section 2.1.1.

To illustrate this, the SPMD code in Figure 2.1-(a) is rewritten as shown in Figure 2.10. The `if` statement in the modified code first checks whether the message with tag `tag1` has arrived. If the message has arrived, it computes `t1` and blocks to receive the message with tag `tag2`. After this message arrives, it computes `t2`. However, the messages may arrive in the reverse order. Therefore, in the case the message `tag1` has not arrived, the `else` part of the `if` repeats the same thing for the reverse order. If both messages have not arrived yet, the code must wait until they arrive (`while` statement) in order to continue with the rest of the code (computation of `t3`). The `while` loop must assure that both `t1` and `t2` have been computed in order to compute `t3`. This requires synchronization variables and/or counters. In this simple dependency case, a flag, `not_done`, is sufficient. However, in general, the synchronization can be quite complicated by the dependencies among the receives and computations. In addition, the complexity of the loop increases with the number of concurrent receives. In general, if there are n receives, then the code must handle $n!$ permutations of receive sequences. Finally, this approach will not take full advantage of message-driven execution since these `if` code-blocks will be scattered around the code and the computations across the `if` code-blocks cannot be executed concurrently. For these reasons, the nested `if` approach is not sufficient to exploit the full power of message-driven execution.

2.4.2 Using a global while-switch loop

A more structured approach to approximate message-driven style is to use a global `while-switch` construct in a module as shown in the Figure 2.11. This approach requires the SPMD program to be decomposed into code-blocks (functions `f1()`, ..., `fn()`) such that these functions can be executed upon receiving a particular message. The processor continuously checks if a message (with any tag) has arrived. Whenever a message arrives, the processor invokes the appropriate code-block, depending on the tag of the message. A simple (and clear) application of this strategy is when the SPMD program can be decomposed into functions such that each one depends on a single message and there are no dependencies among the functions. Otherwise (*i.e.*, in the presence of complex dependencies), the functions or the loop itself have to be augmented with all the synchronization constructs. As a result, the code would not appear as readable as the one in the example. Although the global `while-switch` loop appears to

```

t0 = h()
not_done = TRUE
while (not_done)
    if (probe(tag1)) {
        recv(tag1,a)
        t1 = f(a)
        recv(tag2,b)
        t2 = f(b)
        not_done = FALSE
    }
    else if (probe(tag2)) {
        recv(tag2,b)
        t2 = f(b)
        recv(tag1,a)
        t1 = f(a)
        not_done = FALSE
    }
recv(tag3,c)
t3 = g(t1,t2,c)

```

Figure 2.10: Using nested if's to simulate message-driven execution.

```

while (TRUE) {
    if (probe(any_message_tag))
        switch (message_tag) {
            case tag1 : recv(tag1,m); f1(m);
            case tag2 : recv(tag2,m); f2(m);
            ...
            case tagn : recv(tagn,m); fn(m);
        }
}

```

Figure 2.11: A global while-switch construct to simulate message-driven execution.

be more powerful than the nested `if` approximation, it still has some fundamental limitations. These limitations can be summarized as follows:

Difficulty in supporting multiple modules: Every module must know the entities in other modules to prevent conflicts. Such global entities include the tags of messages defined in different modules. Adding a new message tag requires knowing all of the tags used across the modules, which destroys modularity. The recently proposed MPI message passing interface [36] solves this tag problem by providing the notion of *contexts*. Another name conflict is the function names. Again, the modules must use different names for a multiple module compilation. These two name conflicts destroy modularity in the compiling phase.

Centralized changes: Any change in the modules, such as addition of more messages, may result in modifying the global loop structure. One may try to provide one individualized loop per module to increase modularity. But then, passing the flow of control across modules becomes difficult (the example discussed in Section 2.1.2). When module B receives a message that belongs to module C, then it should call the appropriate function in C. This further complicates a modular design of message-driven programs.

Dependencies must be reflected in the loop: The dependencies among the functions and messages must be handled either inside the function or in the loop, presenting a further programming difficulty.

No message scheduling: The basic `while-switch` loop has no scheduling control strategy. The messages are processed in the order in which the processor provides them to the global loop (which is FIFO).

Difficulty in supporting dynamic objects (computations): Finally, this approach in SPMD does not support dynamic creations of objects (here the functions with their local state).

These limitations prevent the SPMD approach from being a programming model for developing large, efficient parallel applications. A message-driven language seems to be a better method for developing such applications. However, such a pick-and-process message loop is a step in the right direction. In fact, such loops are used in the underlying implementation (or runtime system) of message-driven languages, which are often done on top of an SPMD system.

A programmer, in the absence of a message-driven language, should use such a loop to derive the performance benefits of message-driven execution.

2.5 Charm – a message-driven system

Charm is a C-based portable parallel programming system that consists of a language and a runtime system. The system allows one to define processes or objects and manages dynamic creation of processes with user-selectable load balancing or scheduling strategies. We will discuss the Charm language in some detail because the rest of the work requires some knowledge of Charm. More detailed information on Charm can be found in the manual and various papers on the subject [32, 60, 61].

2.5.1 The Charm language

A Charm program is a collection of processes (called chares) and messages. The chares are potentially small-grained message-driven objects. They have a state and they react to messages with different behavior. At any time, many chares may exist on each processor, or only a few, depending on the application.

2.5.1.1 Chare definition

A chare definition consists of local variable declarations, entry point definitions, and private function definitions as illustrated in Figure 2.12. Local variables of a chare are shared among the chare's entry-points and private functions. Private functions are not visible to other chares and can be called only inside the owner chare. Messages are sent to and received at the entry-points. An entry-point consists of a message definition of certain type and a code-block. The code-block can access the message by the name defined in the message definition.

Some of the important Charm system calls are:

CreateChare(chareName,entryPoint,msg)

This call is used to create an instance of a chare named as `chareName`. As all other Charm system calls, `CreateChare` is a non-blocking call, that is, it immediately returns. Eventually, as the system creates an instance of chare `chareName`, it starts to execute the `entryPoint` with the message `msg`.

```

chare chare-name {
    local variable declarations
    entry EP1 : (message MSGTYPE *msgptr) {C code block}
    ..
    entry EPn : (message MSGTYPE *msgptr) {C code-block}
    private function-1() {C code block}
    ..
    private function-m() {C code block }
}

```

Figure 2.12: Chare definition.

SendMsg(entryPoint,msg,chareID)

This call deposits the message `msg` to be sent to the `entryPoint` of chare instance `chareID`. `chareID` represents an instance of a chare. It is obtained by a system call `MyChareID()`, and it may be passed to other chares in messages.

2.5.1.2 Replicated objects: branch office chares

A branch office chare (BOC) is a form of chare that is replicated on all processors. An instance of a BOC has a branch chare on every processor. A BOC definition is similar to a chare definition (with the `BranchOffice` keyword). In addition to the features of regular chares, a BOC contains public functions which can be called by other chares on the same processor. Also, it can send and broadcast messages to the branches of the same instance with the calls `SendMsgBranch` and `BroadcastMsgBranch`.

2.5.1.3 A Charm example

A simple Charm example is given. The rest of the thesis uses the Charm language, therefore a simple Charm program example will make it easier to understand the other examples.

This example, Figure 2.13, is a branch office chare that implements a ring communication where processor 0 sends a message to 1, then 1 sends it to 2, etc. When processor 0 receives the message, the program terminates. In the figure, only the branch office definition is listed. A Charm program has a main chare in which the execution starts. The branch office ring has

```

BranchOffice ring {
    int next_node;

    entry BranchInit : (message MSG *msg) {
        RING_MSG *ring_msg;
        next_node = (McMyPeNum() + 1) % McMaxPeNum();
        if (McMyPeNum() == 0) {
            ring_msg = (RING_MSG *) CkAllocMsg(RING_MSG);
            SendMsgBranch(loop,ring_msg,next_node); }
        }
    entry loop : (message RING_MSG *msg) {
        if (McMyPeNum() == 0) CkExit();
        else SendMsgBranch(loop,msg,next_node);
    }
}

```

Figure 2.13: A branch office – ring program.

two entries: `BranchInit` and `loop`. The `BranchInit` entry is the creation entry. That is, in some other part of the program, for example `main`, an instance of the ring is created by a `CreateBoc` call

```
ring_instance = CreateBoc(ring,ring@BranchInit,init_msg);
```

which returns a unique identifier for the instance of ring that has been just created. The creation process includes allocation of the local data structures of the branch chare and execution of the entry point specified in the `CreateBoc` call, `BranchInit`, on each processor.

In the `ring` example, when the BOC is created, only processor 0 sends a message to the branch of the same instance on the next processor (directed to the `loop` entry). Then the branch on processor 1 receives the message and passes it to processor 2, and this continues until processor 0 receives the message.

The Charm language has many system calls such as `McMyPeNum()`, `CkAllocMsg` etc. The list and details of these calls and other information can be found in the manual.

2.5.1.4 Modules

One of the strong points of the Charm system is its support for developing modular message-driven programs. A Charm program file consists of a module, which contains names (definition) of messages, specifically shared variables, chares, branched chares, and C functions. These names are internal to the module. Some of these names can be made external and other modules can access these names. A name in another module is referred to by prefixing the name with the module name. For example, in order to access an entry point E of a chare C in a module M, the prefixed name would be `M:CE`.

Without entering in the details, we will give an example of multiple modules because it is relevant to our discussion in this section and in the following sections. A module uses an `interface` construct to declare its exported or imported names. Figure 2.14 shows an example interface construct. The module M2 imports some entities from the module M1 by including the interface file of M1 — `M1.interface`. The interface file defines the entities such as types, chares, and entry-points that are allowed to be accessed from other modules. In this example, M2 creates an instance of a chare, `chare1`, defined in M1 by accessing the chare name and entry point name, and message type. Similarly, other entities can be accessed.

2.5.1.5 Other language features

In addition to messages, Charm provides other ways in which processes (chares and BOCs) share information. The discussion of these features is not closely relevant to the discussion here. The details about these mechanisms — namely, `readonly`, `writeonce`, `monotonic variables`, `accumulators`, and `distributed tables` — can be found in [16].

2.5.2 The Charm runtime

The system keeps a pool of messages on each processor. These messages are either for creation of new objects (`CreateChare`) or for existing objects (`SendMsg`). Each message contains the instance of the chare and the entry point in addition to user data. On each processor, the runtime system repeatedly selects one of the available messages from the pool and executes the entry point code indicated by the message. This execution may change the state of the chare instance and may deposit new messages into the message pool. On a particular processor, an

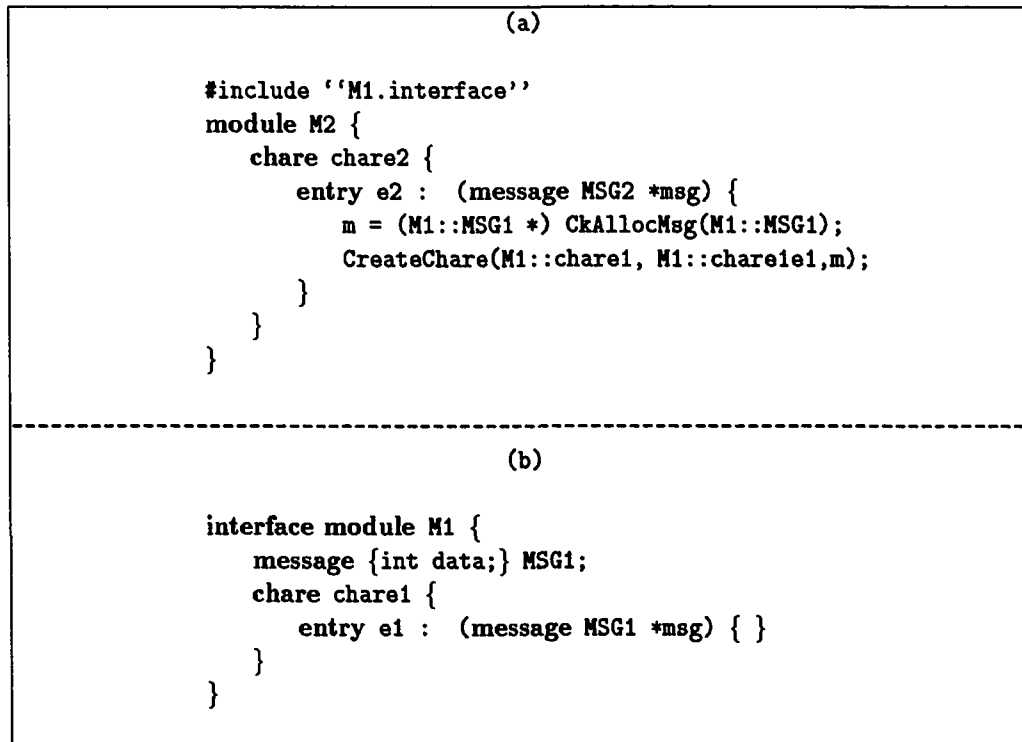


Figure 2.14: Multiple modules in Charm (a) module M2 accesses entities in module M1 (b) interface module for M1 M1.interface.

entry point executes until completion and is not interrupted for activating another entry point code.

The message scheduling – selection of the messages from the pool – is done according to the scheduler. The system provides FIFO, LIFO, and priority-based scheduling policies, from which users can select one that suits their application. In addition, they can be substituted with other user supplied strategies.

2.6 Related work on latency tolerance

The idea of *multiple contexts* in which a processor switches between contexts in order to hide latencies is used at various levels in computer systems — from the language level to the architecture level. An early work in this area includes the *future* construct of [49]. A future is a process that can be executed in parallel with its parent process. When the parent needs the

value of the future, which is produced by the child, it blocks until the child has completed. This mechanism is analogous in some ways to SPMD style: if creation of a future is considered analogous to a request to some remote processor, then touching the future is equivalent to a blocking receive. This is not fully adaptive, because when waiting on multiple futures, the caller blocks for them in a fixed sequence. A future based system achieves a degree of adaptive overlap, as multiple futures are assigned to the same processor concurrently assuming that a mechanism is provided for switching between them (by using threads, for instance).

The original Actor model as described in [1] is purely message driven. Actors are self-contained, interactive, independent components of a computation that communicate by asynchronous messages. Chares are similar to actors. One difference is that the methods of an actor can be executed concurrently (as defined in the Actor model), but the entry-points of a chore cannot be executed concurrently. One of the first implementations of the Actor model to run on parallel computers, HAL [56], was implemented on top of Charm. Other concurrent object oriented languages include ABCL/1 [6] and Concurrent Aggregates (CA) [17]. In ABCL/1, objects process one message at a time also, and objects can selectively receive messages. The CA language supports fine grain parallelism. An aggregate is a collection of objects that has a single name. It supports concurrency within objects which can be distributed across processors. An object can create threads of computations by using the `conc` construct (similar to `parbegin` and `parend`), and the variable synchronization is done by context futures.

The performance of message-driven execution can be improved significantly with some operating system support. Active Messages [31] is a recent work in this respect. Active Messages itself is not a message-driven model, but it provides the operating system support to build message-driven systems efficiently. It notifies computations of the arrival of a message by invoking a user-defined message-specific handler. Thus, properly implemented, it provides an inexpensive user-level interrupt. The split-C language based on Active Messages employs polling for arrival of messages [20].

The Reactive Kernel/Cosmic Environment developed by Seitz, et al. [5] provides a mechanism for switching context based on the next available message. It invokes an appropriate handler depending on the message received and in this respect it anticipates the functionality of Active Messages. However, scheduling is not interrupt driven; context switching is done only when one process blocks for a message.

At the architecture level, multithreading is used to hide memory access latencies. When a thread is forced to wait for a remote memory access, the processor can switch to another thread. Multithreading generally deals with fine grain computations. Multithreading yields improved performance by overlapping communication latency with computation [9, 71]. However, threads allow no (or very limited) control over scheduling of messages. Also, a single thread cannot block for multiple data items at once (*i.e.*, no split phase transactions). Creating another thread for such cases might be expensive. If scheduling of threads is based on time-slicing then locking is necessary to ensure the correctness of shared data which will add some overhead and complexity to the program. The first multithreaded machine was HEP [87]. The TERA machine, a descendant of HEP, [3] supports very fast context switches between threads, and activation of threads are based on arrival of data from global memory. Other fine grain multiple-context architectures include the MIT J-Machine [22] and Mosaic [5]. These are examples of message-driven architectures. The basic building block of the J-Machine is its message-driven processor. The processor creates a task for each arriving message, and messages are scheduled with a FIFO policy.

Chapter 3

Two obstacles

As shown in the previous chapter, message-driven execution has potential performance benefits over the traditional SPMD model, especially in developing large and multiple-module parallel applications efficiently. In order to quantify and study these benefits, we want to write message-driven programs and conduct performance studies. However, developing such programs and conducting the proposed performance studies in a pure message-driven language such as Charm is complicated due to the split-phase transactions necessitated by message-driven execution and the nondeterministic arrival of messages, which requires complex reasoning about the concurrent activities in such programs. In this chapter, these difficulties are identified, which then motivate the work done in Chapter 4 and 5.

3.1 Programming difficulties of message-driven style

In order to illustrate the problems encountered in writing message-driven programs, we will use the SPMD program presented in Figure 2.1 in Chapter 2. A typical SPMD program, like this one, contains a sequence of blocking receive points and code-blocks, which are sequences of instructions that can run without any remote data access. Writing this code in a message-driven language requires splitting the code into these code-blocks. We will call the code-blocks entry-functions (following the terminology in Charm).

The operation of accessing remote information is split into two parts that become physically separated in the program:

- sending the request to fetch the information,

- the code-block to handle the data when they arrive.

Without loss of generality, we assume that each receive statement and its dependent code-block becomes an entry-function. As a result, the SPMD program, which is a single entity with a single thread of control, is transformed into a message-driven program as a collection of many entities (entry-functions).

A message-driven program (in Charm) for this SPMD example is listed in Figure 3.1. In a pure message-driven model, an entry-function is immediately scheduled for execution when its associated message has arrived. However, often there is a dependence relation among these code-blocks. For example, in this program, the calculation of `t3` cannot be performed until `t1` and `t2` are completed. Hence, the entry-functions must contain additional code, written by the programmer, to maintain such dependences. The message-driven program in Figure 3.1 does not perform any dependence check. It thus implicitly assumes that the messages arrive in the proper order (i.e., the entry-function `tag3` is executed last). If messages arrive in a different order, then this code will execute incorrectly (in practice, if `tag3` arrives before either `tag1` or `tag2`). Therefore, message-driven programs must consider the arrival order of messages. This synchronization dependence in the user code complicates the message-driven programs, particularly if the order of message arrivals is unpredictable.

3.1.1 Nondeterministic message arrival

The order of message arrivals depends on many runtime conditions. Messages might originate from different sources, and delay in the creation of messages at the source is one of the possibilities that may change the message sequence. Even the sequence of messages sent by the same source might be different at the destination on some machines because they may follow different paths. In addition to delays in the network, the message scheduler can change the order of messages to be processed. In short, the order of messages processed may change dynamically and is unpredictable. Therefore, a message-driven program must handle different sequences of messages.

In order to adhere to the dependences for correct execution, the message-driven entry-functions must ensure that the code-blocks that it depends on are completed. The implementation of this synchronization mechanism necessitates usage of shared control data (such as

```

chare incorrect {
int t1,t2,t3;
  entry init : { t0 = h(); }
  entry tag1 : (message MSG *a) {
    t1 = f(a);
    CkFreeMsg(a);
  }
  entry tag2 : (message MSG *b) {
    t2 = f(b);
    CkFreeMsg(b);
  }
  entry tag3 : (message MSG *c) {
    t3 = g(t1,t2,c);
    CkFreeMsg(c);
  }
}

```

Figure 3.1: Incorrect message-driven code.

counts, flags and buffers) among the entry-functions and depends on the atomicity of the entry function.

A correct message-driven implementation of the SPMD example program in Charm is listed in Figure 3.2 with the required synchronization mechanism. The required data structure for this simple example is only a counter (*count*) and a message buffer *buffer*. These variables are declared as local to the chare so that the entry-functions can share them (not at the same time, since only one of the entries is executing at a given time). Assume the execution of the program is started at the *init* entry-function (there always exists such an entry-function for every chare in Charm programs). This guarantees that the other messages to the other entry-functions will not be delivered until completion of the *init*. The entry-function *tag1* and *tag2* can calculate *t1* and *t2* respectively without any dependence check whenever they are activated. The entry-function *tag3*, on the other hand, must buffer its message if it detects that at least one of *tag1* and *tag2* is not completed yet. This is accomplished by the *count* variable. In the *init*, we set *count* to the value 3, which is the number of inter-dependent entry-functions. Each entry-function decrements this counter by one when it is activated. If an entry-function

```

chare correct {
int count;
int t1,t2,t3;
MSG *buffer;

    entry init : {
        t0 = h();
        count = 3;
    }

    entry tag1 : (message MSG *a) {
        t1 = f(a);
        count = count-1;
        if (count == 0) {
            t3 = g(t1,t2,buffer);
            CkFreeMsg(buffer);
        }
        CkFreeMsg(a);
    }

    entry tag2 : (message MSG *b) {
        t2 = f(b);
        count = count-1;
        if (count == 0) {
            t3 = g(t1,t2,buffer);
            CkFreeMsg(buffer);
        }
        CkFreeMsg(b);
    }

    entry tag3 : (message MSG *c) {
        count = count-1;
        if (count == 0) {
            t3 = g(t1,t2,c);
            CkFreeMsg(c);
        }
        else
            buffer = c;
    }
}

```

Figure 3.2: Correct message-driven code.

detects that the counter is zero, then it knows it is the last one. The entry `tag3` calculates `t3` if the count is zero (*i.e.*, others have been completed). Otherwise, it buffers its message. Since the count is non-zero, at least one of `tag1` and `tag2` must be activated eventually. The one that is activated last detects this and knows that `tag3` buffered its message. Therefore, after completing its calculation, it fetches the message `c` and computes `t3` on behalf of the entry-function `tag3`.

This example illustrates how complicated message-driven programs can become even with a simple dependence relation. Therefore, a message-driven language must provide necessary language support to simplify this.

3.1.2 Obscure flow of control

Typical traditional SPMD programs, like the one considered in this chapter, exhibit a relatively clear flow of control. The flow of control is the sequence of instructions executed by the program. Since SPMD imposes a total order on the code-blocks, the flow of control is easily identifiable in such programs.

The actual dependences usually result in a *partial order* on the code-blocks, and message-driven style exploits this fact. However, the flow of control for this partial order is not easily visible in a pure message-driven program text or representation. As can be observed in the message-driven program written in the previous section, it is difficult to identify the dependences among the code-blocks due to complex synchronization mechanisms, including conditional executions (such as that of `t3` based on `count`).

From the programmer's point of view, the difference among these models can be visualized as in Figure 3.3. Part (a) and (b) depict how SPMD and pure message-driven programs are perceived by the programmer, whereas the dependences among the code-blocks are shown in part (c). These figures pictorially motivate why it would be useful to have a clear representation of dependences (as in part (c)) in message-driven programs. Chapter 4 describes a coordination language, Dagger, which is essentially based on this observation.

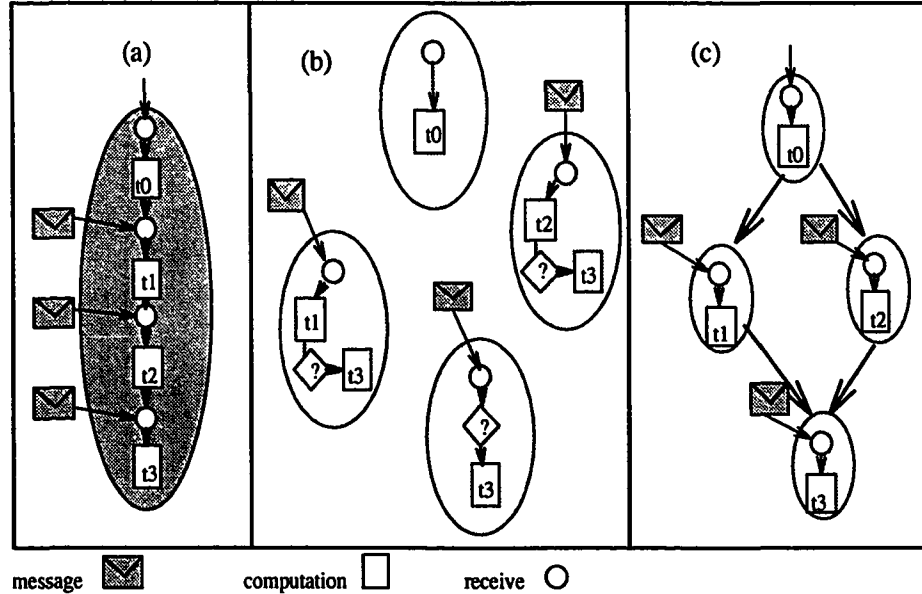


Figure 3.3: Flow of control (a) SPMD (b) pure message-driven (c) partial order.

3.2 Performance prediction: how to simulate message-driven computations

Another issue is how to predict the performance of message-driven programs. Simulations (particularly trace-driven simulations, see Chapter 5) are commonly used for predicting performance of computer systems. For reasons given in Chapter 5, we have chosen the trace-driven simulation approach for performance analysis.

However, the nondeterministic behavior of message-driven programs makes their simulations difficult. We will illustrate the difficulty by using the same example we used in the previous sections.

Assuming the entry `init` is executed at the creation of the process, the process awaits two messages concurrently. In a particular execution, assume that the message directed at the entry point `tag1` arrives first. This will cause the execution of the code $t_1=f(a)$. Then, when the message for `tag2` arrives, the processor executes $t_2=f(b)$ followed by $t_3=g(t_1,t_2,c)$. However, if the messages arrive in the reverse order, the code at `tag2` would execute only $t_2=f(b)$, leaving the execution of $t_3=g(t_1,t_2,c)$ to the other entry point.

An accurate trace-driven simulation of this program is not possible, as seen below, without a complex dependence analysis of various paths through each entry point. A normal trace will consist of the duration of execution of each entry point (and relative timings of any message sent during it). If traces are obtained with the former sequence with A time units for the execution of tag1, and B time units for tag2, and during simulation the machine conditions lead to the latter sequence, it is not possible to reconstruct the times of tag2 and tag1.

If the individual times of calculations t1, t2 and t3 are recorded, one would be able to reconstruct the timings in presence of the new sequence. It may seem simple then to record these times. However, note that f(a), f(b) and g(t1,t2,c) need not be function calls as shown here. The if statements, as well as the computation blocks, might be deeply buried inside complex control structures. Therefore, in general, it is not easy to retrieve the timings of the individual blocks. Furthermore, the connection between the value of the counter becoming zero and the arrival of messages may not be easy for the compiler to deduce.

Both of these obstacles, programming difficulties and performance prediction difficulties, will be overcome using the Dagger notation described in the next chapter.

Chapter 4

Controlling the complexity of message-driven programs

As we discussed in earlier chapters, message-driven execution has the potential for yielding substantial performance improvements in parallel programs. However, we also saw that message-driven programs suffer from a lack of clarity of expression (resulting from code) to deal with internal synchronization, buffering of messages, the conditional execution of code based on flags, etc. This can also lead to message-ordering bugs. If message-driven execution is to be widely used or adopted by the parallel programming community, it must be supported with better programming notations to eliminate these difficulties.

As argued in the introductory chapter, divide-and-conquer is a paradigm that is naturally suited for message-driven execution. Divide-and-conquer computations can be programmed in a language such as Charm. However, Charm is a general purpose language. For this special application domain, one can customize and simplify the expression of message-driven parallelism by providing certain commonly needed features automatically. This was accomplished by developing a divide-and-conquer sublanguage [44] on top of Charm that is described in Section 4.1.

Based on the experience obtained while designing the divide-and-conquer sublanguage, we then set out to design a general purpose message-driven language that deals with common difficulties encountered in pure message-driven languages, such as Charm. In particular, as shown in Chapter 3, these difficulties include the lack of clarity of flow of control, an increased

likelihood of message-ordering dependent bugs, and the need to deal with various possible orderings by using combinations of buffering, flags, conditionals, and reasoning about plausible and implausible message orderings. The resulting Dagger language [47], which is the central part of this chapter, is described in Section 4.2. The features of this language are also useful for carrying out accurate simulation of message-driven programs as shown in Chapter 5.

In traditional SPMD programs, there is a natural disincentive for developing an application modularly as a collection of separately usable modules, because of the difficulty of overlapping communication and computations across modules as shown in Chapter 2. As message-driven systems remove this obstacle, programmers are likely to develop applications as a collection of many modules, each of which can be used in a different context. The techniques involved in interfacing various modules in a message-driven context are described in Section 4.3.

4.1 Divide-And-Conquer: a simpler context

In this section the divide-and-conquer coordination language, which is a preliminary step toward the development of Dagger language, will be described briefly. More details of the language are given in [44].

Divide-and-conquer is a naturally parallel paradigm that is broadly applicable [2, 73]. In a typical divide-and-conquer computation, a computational problem is broken down into smaller subproblems, some of which may be of the same type as the original problem, but of lesser complexity. This process is continued recursively as many times as necessary. When the subcomputations are simple enough they are solved directly, without further subdivision. The results from the subcomputations are passed to the parents that created them. The parent node combines the solutions to subproblems to form the solution to itself, which it then sends to its parent.

Variations on this theme are also possible within the paradigm. In search-type problems the combination of subproblems is either trivial or absent (if solutions are directly printed). In some other domains, solutions to some subproblems may lead to creation of new subproblems that must be solved. This can happen, for example, due to a data or control dependency among the subproblems.

Many problems, such as combinatorial optimizations, searches [67], many problems in computational geometry, numerical methods [39, 75], and problem-reduction in AI, are formulated naturally as divide-and-conquer computations. Divide-and-conquer is identified as an important strategy for higher level image processing algorithms in [89], and the significance of software support for expressing parallelism is stressed. An implementation of a software package that supports tree computations has been discussed in [34]. It is a library of services that are linked to user-supplied code. Similar approaches have been discussed in [12]. In [38], the implementation of a software system that supports a divide-and-conquer paradigm is discussed. Its implementation on nonshared memory machines has not been completed yet, and the author states that it is a challenging task. Also, neither approach supports arbitrary dependencies among subproblems.

4.1.1 Language definition

In order to develop this language, we observe that a typical divide-and-conquer computation needs to fire its subcomputations and express dependences between results of subcomputations (each subproblem returns one solution) so that, when results of certain of subcomputations are done, it can carry out some local computations and fire new subproblems; and it must provide a way of communicating the inputs to the subproblem and the outputs from the subproblems.

These abilities are provided by allowing a definition of a node, Figure 4.1, to include a set of input parameters, a set of output parameters, statements for firing subproblems, and statements for expressing synchronization of dependences of the results of subproblems. These are provided in the form of `in`, `out`, `fire`, and `when-block` constructs of the divide-and-conquer language.

4.1.2 Data declarations

in : { *declaration-list* }

Specifies the formal parameters to be received by value.

out : { *declaration-list* }

Specifies the formal parameters to be sent to the parent instance by value.

node *node-declaration-list*

Each subcomputation must have a distinct label to differentiate it from others. *node-*

```

node node-name {
  in : { variable-declarations }
  out: { variable-declarations }
  node node-names-list;
  cond cond-name-list;
  local-variable-declaration
  init : { init-body }
  when condition-list : { when-body }
}

```

Figure 4.1: Divide-and-conquer node definition.

declaration-list declares all labels used in the node, and specifies the node type that each label refers to. Input and output parameters of a subcomputation are accessed through the pointers *label.in* and *label.out*, respectively. A node can access its own input and output values through the pointers *in* and *out*.

cond *cond-declaration-list*

Any *condition* variable that is used in the *condition-list*'s of *when-block*'s or in the *set* statements (see below) must be declared here.

4.1.3 Blocks

init : { *init-body* }

When an instance of a node is created, its *init-body* code is executed. This code initializes the node instance and usually performs the termination check. It spawns new subcomputations if it decides to divide the problem further, otherwise it solves the problem directly. After *init-body* is completed, the node instance suspends itself until one of its *when-block*'s becomes eligible for execution.

when *condition-list* : { *when-body* }

A *when-block* is a synchronization point where the solutions from subcomputations are combined. The labels in the *condition-list* refer to subcomputations or condition variables. If solutions from all of the subcomputations listed in the *condition-list* are available and

the condition variables listed in the *condition-list* have been set (by the **set** instruction), then the code *when-body* is scheduled for execution. If more than one *when-block*'s are satisfied, they are executed one after the other with no particular order.

4.1.4 Statements

fire *label*

Creates an instance of the node that is associated with the node *label*. The input parameters of the node that is to be created, *label.in*, must be initialized before invoking the **fire** statement. With the execution of the **fire** statement, the control of the data area pointed to by *label.in* is transferred to the subcomputation, and it must not be accessed subsequently. Similarly, the data pointed by *label.out* becomes valid only after the subcomputation *label* has been completed.

send result

A node sends its solution (pointed by **out**) to its parent node with this statement. In addition, the memory space allocated to all responses that are received from subcomputations are released, and execution of the node instance is terminated.

4.1.5 An example

A simple example, Fibonacci numbers, is listed in Figure 4.2. A node, **fib**, is defined to calculate the computation $f(n) = f(n - 1) + f(n - 2)$ recursively. Assume that an instance of node **fib(n)** is created. Then, this node creates two child-nodes: **fib(n-1)** and **fib(n-2)**. In order to do that, it declares two node labels of its own type, **p** and **q**. The creation of subcomputations are done by firing **p** and **q**. After **p** and **q** completes, they return solutions to their parent simply by the statement **send result** (without specifying the details such as parent's processor number, etc). The divide-and-conquer system automatically executes the *when-block* - **when p,q** - of the parent node after the solutions from both children have been received.

As illustrated by this example, parallel implementation of divide-and-conquer algorithms is significantly simplified by the node construct, and the user is freed from the following laborious tasks:

```

node fib {
  in : { int n; }
  out: {int result;}
  node fib : p, fib : q;

  init : {
    if (in->n < 2) {
      out->result = in->n;
      send result;
    }
    else {
      p.in->n = in->n - 1;
      q.in->n = in->n - 2;
      fire p;
      fire q;
    }
  }
  when p,q : {
    out->result = p.out->result+q.out->result;
    send result;
  }
}

```

Figure 4.2: Node definition to compute Fibonacci numbers.

- synchronization management : keeping track of responses from subcomputations and execution of *when-block*'s if their conditions are met.
- tree communication : handling parent-child communication.
- allocation : automatic allocation and deallocation of messages.
- dynamic load balancing.
- machine-dependent expression.

4.2 Dagger

Although the divide-and-conquer language serves for the purpose for which it was designed, it has narrow applicability. It relies heavily on the fact that every subproblem fired returns exactly one solution and uses this information to eliminate explicit sending and allocation of messages. Divide-and-conquer computations also have a causal connection between the firing of a subproblem and the receipt of the solution for it, i.e., the solution of a subproblem cannot arrive before the subproblem is fired. In the general case, such assumptions are not valid, and a more general language is needed. We now describe Dagger, a more general purpose language [47], that retains the performance benefits of message-driven execution while capturing some of the clarity of SPMD programming. The language has been implemented in the Charm parallel programming system, and Dagger programs run portably on a variety of parallel machines.

4.2.1 Expecting a message

In divide-and-conquer, we used the *when-block* construct to synchronize with the arrival of more than one message. In Dagger, we retain the *when-block* concept as well as augment the language with the notion of *expecting* a message. In Charm, an entry point is executed when there is a message directed to it. If the computation in that entry point depends on computations in the other entry points within the same chore that are not executed yet, then the programmer must handle this unexpected message by buffering it and later fetching it whenever the entry point becomes eligible for execution. In Dagger, on the other hand, a computation is triggered by the availability of expected messages as illustrated in Figure 4.3.

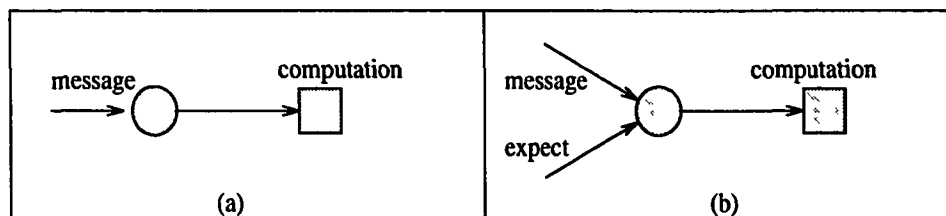


Figure 4.3: A message triggers a computation (a) in pure message-driven (b) in Dagger.

With *expect* and *when-block* constructs, Dagger allows us to specify processes in terms of dependences between messages and pieces of computations. We will explain by examples

how expect and when-block constructs are used to express such dependences along with the description of the language. Before presenting the details, however, we will start with a simple example to introduce the *dag-chare* that represents a process (or chare) in Dagger.

4.2.2 An example

Consider an algorithm for matrix multiplication that uses dynamic load balancing. Such a formulation may be useful on a machine where different processors operate at different speeds, for example. We assume that the two matrices to be multiplied have been stored in distributed tables¹. Matrix A is stored as a collection of entries such that each entry is a block of contiguous rows. Similarly, matrix B is stored as a collection of columns.

First, we will describe the Charm implementation of the matrix multiplication. One of the chares (*mult_chare*) used in implementing such an algorithm is shown in Figure 4.4. This chare is responsible for multiplying a block of rows of A and a block of columns of B. The entry *init* is executed when an instance of the chare is created. The message *msg* contains indices of the row and column blocks that are to be multiplied.

When an instance of *mult-chare* is created, it requests the row and columns from *Atable* and *Btable* (these tables store the matrices A and B) by calling *Find*, which is a system call in Charm. In the *Find* call, the row (or column) index, return entry-point, and the chare instance identifier are supplied. Note that *Find* is non-blocking, and it immediately returns. Eventually, the row and the column data will be sent in a message (of type *TBL_MSG* that is defined in the Charm language) to the entry-point *recv_row* and *recv_column*, respectively.

The multiplication depends on the availability of both rows and columns. The messages for rows and columns may arrive in any order. Therefore, two variables (shared between entry-points): a counter, *count*, and a message buffer, *buffer*, are utilized. *count*, set to two initially, is used to ensure that both messages arrived before calling *multiply*. When the first message is received, it is saved in *buffer* (because it can not be used unless the other one is also available), and *count* is decremented by one. When the second message is received, *count* becomes zero, and *multiply* is called with the buffered message and the current message.

¹Distributed tables are one of the information sharing mechanisms supported by Charm (hence by Dagger) where data are distributed across processors and accessed asynchronously.

```

chare mult_chare {

  int count;
  ChareIDType chareid;
  TBL_MSG *buffer;

  entry init : (message MSG *msg) {
    count = 2;
    MyChareID(&chareid);
    Find(Atable, msg->row_index, recv_row, &chareid, TBL_NEVER_WAIT);
    Find(Btable, msg->column_index, recv_column, &chareid, TBL_NEVER_WAIT);
  }

  entry recv_row : (message TBL_MSG *row) {
    if (--count == 0 )
      multiply(row->data, buffer->data);
    else
      buffer = row;

  entry recv_column : (message TBL_MSG *column) {
    if (--count == 0 )
      multiply(buffer->data, column->data);
    else
      buffer = column;
  }
}

```

Figure 4.4: Matrix multiplication chare.

Even in this simple example, the handling of messages for correct execution complicates the code. In the next section, the same algorithm will be presented in Dagger notation.

4.2.2.1 Example with a dag-chare

The Dagger language is defined by augmenting Charm with a special form of chare called a *dag-chare*. A dag-chare specifies pieces of computations (when-blocks) and dependences among computations and messages. A when-block is guarded by some dependences that must be satisfied before the when-block can be scheduled for execution. These dependences include

```

dag chare mult_chare {

    entry init : (message MSG *msg);
    entry recv_row : (message TBL_MSG *row);
    entry recv_column : (message TBL_MSG *column);

    when init : {
        MyChareID(&chareid);
        Find(Atable, msg->row_index, recv_row, &chareid,TBL_NEVER_WAIT);
        Find(Btable, msg->column_index,recv_column,&chareid,TBL_NEVER_WAIT);
        expect(recv_row);
        expect(recv_column);
    }

    when recv_row, recv_column : { multiply(row->data,column->data) }
}

```

Figure 4.5: Matrix multiplication dag-chare.

arrival of messages and completion of other when-blocks. Before describing the Dagger language in detail, let us consider the matrix multiplication example and show how it looks in Dagger.

Figure 4.5 shows the matrix multiplication written as a dag-chare. The dag-chare declares two entry-points for the row and the column messages as well as a initialization entry `init` (the `init` entry has a special meaning in Dagger and it will be discussed later). The when-block `when init` is executed when an instance of `mult_chare` is created. This when-block makes the table requests and expects the messages for rows and columns, and the dag-chare suspends its execution. After both of the row and the column messages received, the dag-chare resumes its execution, and `multiply` is called (in the second when-block). Note that the messages in the `multiply` call are accessed by the variables defined in the corresponding entry-point declarations.

Dagger handles the necessary bookkeeping, such as counters, flags and message buffers. Therefore, the resulting code is more readable (and easy to program), while still retaining the benefits of a message-driven model.

In this simple example, the column and row messages cannot arrive before the invocation of `Find` statements. This time-dependency is similar to the one in the divide-and-conquer model.


```

dag chare template {

    local variable declarations
    condition variable declarations
    entry declarations

    when depn_list_1 : when_body_1
    ...
    when depn_list_n : when_body_n

    private function f1()
    ...
    private function fm()
}

```

Figure 4.6: Dag-chare template.

Therefore, the **expect** statements might seem redundant. The purpose of this simple example was to introduce the basic structure of the dag-chare. In the following sections, other examples will illustrate the necessity of **expect**.

4.2.3 Basic language

A process or object is defined by the dag-chare construct. A template for the dag-chare is shown in Figure 4.6. The dag-chare declares entry-points as well as some other data in the local variable declaration section. The local variables are shared among when-blocks and private functions of the dag-chare. Private functions are regular C functions that may contain Charm or Dagger statements/calls, and they can be called only within the static scope of the dag-chare.

4.2.3.1 Entry points

An entry-point declaration is in the form:

```
entry entry_name : (message msg_type *msg)
```

It defines an entry with the name `entry_name` and associates a variable and a message type with it. The variable `msg` is a pointer to the message received by the entry. The message

received at the entry point is accessed through this variable name. Messages can be sent to entry points as in Charm by supplying `entry_name` in the Charm system calls such as `SendMsg`. The basic difference between a dag-chare entry and chare entry is that a dag-chare entry does not associate an action (entry-function) with the entry. Each dag-chare must declare an entry point called `init`. Messages for this entry are expected implicitly. An instance of the dag-chare is created by sending a message to this entry (by `CreateChare` call), and the corresponding when-block is executed upon creation of the instance.

4.2.3.2 Expect statement

Receiving a message at an entry point is not sufficient to trigger a computation. A Dagger program tells the Dagger runtime system when it is ready to process a message by using the **expect** statement:

```
expect(entry_name)
```

If a message arrives before an **expect** statement has been issued for it, Dagger will buffer it. The message becomes available only after the **expect** statement is executed.

4.2.3.3 Ready statement

A dag-chare may have a special type of variable called *condition-variable*. A condition-variable is declared as follows:

```
CONDVAR cond_var_name
```

Condition variables are used to signal completion of when-blocks. In other words, they express the dependences among when-blocks that belong to the same dag-chare instance. The **ready** statement is the combination of send and **expect** operations that involves the entries within the same dag-chare. A when-block can send a message to an entry that is defined in the same dag-chare. However utilizing a shared variable (condition-variable) is more efficient for this purpose. A condition-variable is initialized to the *not-ready* state when it is declared. It is set to the *ready* state by the **ready** statement:

```
ready(cond_var_name)
```

Once a condition-variable is set, Dagger may schedule the when-blocks that are waiting for that condition-variable to be set.

4.2.3.4 When-blocks

A when-block is a computation which is guarded by a list of entry names and condition-variables.

when $e_1, \dots, e_n, c_1, \dots, c_m$: **when-body**

where e_i is an entry name and c_i is a condition-variable. In order to initiate the execution of a when-block, the dependence list of the when-block must be satisfied. The dependence list is satisfied if :

- a message has been received and **expect** statement has been executed for each entry e_i in the dependence list,
- a **ready** statement has been executed for each condition-variable c_i in the list.

The **when-body** is a block of C code possibly including Charm system calls, and **expect** and **ready** Dagger statements. Messages received by the entries e_i 's are accessed inside the **when-body** through the message pointers defined in the entry declarations of the corresponding messages.

4.2.4 Dag-chare example

The example discussed in Chapter 2 and 3 (its Charm code appears in Figure 3.2) is coded in Dagger in Figure 4.7. As one can see, the code retains the adaptability of the Charm code. The computation of t_1 or t_2 will precede, depending on which message arrives first, and t_3 is calculated only after both t_1 and t_2 have been completed. However, the flow of control is much more apparent than the one in the corresponding Charm program: the details of synchronization is hidden from user.

4.2.5 Extended language

The Dagger language as defined so far was kept simple for ease of exposition. Supporting the full generality of parallel programming requires two extensions embodied in Dagger, which are motivated and described below.

The **expect** statement imposes an order on the execution of messages. This is sufficient for a simple dag computation (a computation where there are no cycles in the dependency

```

dag chare example1 {
    int t0,t1,t2,t3;

    CONDVAR t1_done, t2_done;
    entry init : (message MSG *m);
    entry tag1 : (message MSG *a);
    entry tag2 : (message MSG *b);
    entry tag3 : (message MSG *c);

    when init : {
        t0 = h();
        expect(tag1);
        expect(tag2);
        expect(tag3);
    }

    when tag1 : {
        t1 = f(a);
        ready(t1_done);
    }

    when tag2 : {
        t2 = f(b);
        ready(t1_done);
    }

    when t1_done,t2_done, tag3 : { t3 = g(t1,t2,c); }
}

```

Figure 4.7: Dag-chare illustrating adaptive overlapping.

graph). However, there are computations where concurrent phases of a dag exist (in time). An example is that of a dag augmented with a loop where different iterations of the loop may be executed concurrently. Another example is a client-server type of computation. Client processes may send multiple requests concurrently to a server dag. The server dag performs the same computation for different requests concurrently.

In the following sections, we will first demonstrate how loops can be represented in Dagger notation. Then we will show the necessity of *reference numbers*, a way of distinguishing messages belonging to different phases of a dag-chare. An important application of reference numbers is pipelining independent iterations of a loop which is common in many parallel numerical algorithms.

4.2.6 Expressing loops in dag-chare

The dag-chare has been used to express directed acyclic dependences so far. Cyclic dependences can also be expressed in Dagger notation by folding the dependency graph dynamically. Cyclic dependences arise in algorithms that contain loops, such as numerical iterative methods. We will illustrate how Dagger supports such computations by presenting examples of some iterative methods to solve a penta-diagonal linear system (which arises in the solution of partial differential equations). We will present the problem without getting into the details of the application, and present the Dagger code for it. This problem involves a 2-dimensional grid of points. The grid is partitioned into rectangular blocks and each processor is assigned one block. Each processor performs some local computation on its own block and exchanges some information (boundary values) with four neighbor processors (east, west, north, south). The computation continues in this manner until a fixed number of iterations or until the solution is reached. To determine if the solution has been reached or not usually requires a convergence check, which is a global operation (reduction). In the following examples, the convergence checks (reductions) are eliminated to simplify the discussion of loops. The reduction operation will be discussed in Section 4.3 with message-driven libraries.

There are a number of iterative schemes that can be used. Jacobi and Gauss-Seidel iterative methods are among the most commonly used ones. For our purpose, we will explain only the computation and communication structure of these algorithms.

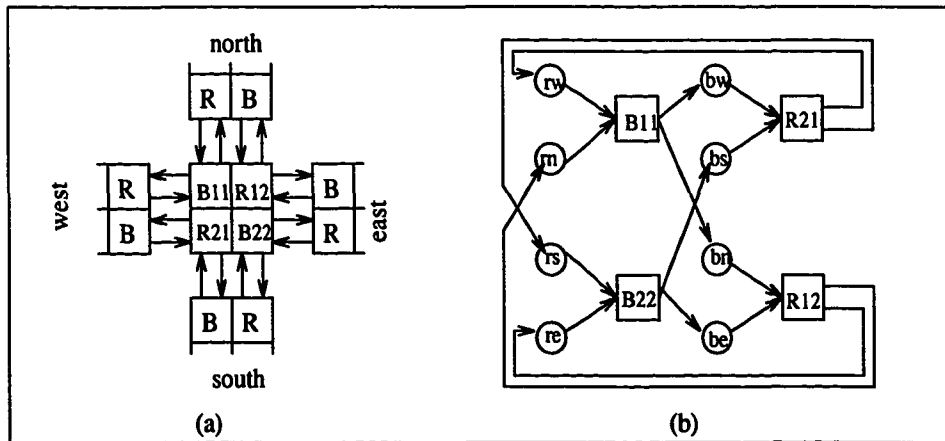


Figure 4.8: Red-black Gauss-Seidel (a) partitions (b) dependences on one processor.

4.2.6.1 Loop example: Gauss-Seidel red-black

The Gauss-Seidel iterative scheme has been widely used in sequential and parallel applications. The Gauss-Seidel scheme with the natural ordering of grid points cannot be parallelized efficiently. Instead, reordering of the grid points, which is known as red-black ordering, results in an easy and efficient parallelization [75]. The basic computation in the red-black scheme is as follows: each processor holds a subset of the grid points. The points are colored as black and red as shown in Figure 4.8-(a). A sweep of grid points consists of two steps: updating the black points, and updating the red points. The dependences (and flow of control) in the computation on one processor are illustrated in Figure 4.8-(b). First, a processor sends and receives the red boundary points from its neighbors. Then it updates its black points B11 and B22 using the red boundaries. Next, it sends and receives the black boundaries that have been just updated. Finally, the processor updates its red points and computation continues until the iteration limit is reached.

The Dagger program for this computation is listed in Figure 4.9. For simplicity, we present only the dag-chare part of the program with necessary details for expressing this computation. The dag-chare `red_black` is a branched chare. A copy of this branch is executed on every processor. The chare declares eight entry points for receiving the boundary points from its neighbors, the four red and four black boundaries, as well as the initialization entry `init`. When

the chare instance is created, the when-block `init` performs initializations, such as determining neighbors, creating the grid points etc., sends the initial red boundaries to its neighbors, and issues `expect`'s for the red messages from its neighbors. Each quarter of the local grid (B11, B22, R12, and R22) depends on two messages. When those messages become available (and they are expected), the points in that quarter are updated. Therefore, four when-blocks are defined (one for each quarter) for sweeping all the grid points. Note that B11 and B22 can be updated in any order followed by R12 and R21 again with any order in a sweep.

The looping is accomplished by expecting the red boundaries again after the calculation of red points (both R12 and R21). The correct order of updating black and red points are guaranteed by the `expect` statements. This program continues until the when-blocks issue no more expects for the red points.

4.2.6.2 Loop example- Jacobi iterations

In this section we will present a similar computation, using the Jacobi method, without convergence checks again, to illustrate that the `expect` statement is not always sufficient to support the correct execution of loops. This computation (the computational structure is depicted in Figure 4.10) is an iterative method similar to the red-black scheme. However, the points are not divided into groups, but all have the same color. At a particular iteration, a processor sends its boundaries to its neighbors and receives four boundary messages from them. Then it updates its local points using the boundary values. The computation continues with the next iteration in this manner.

The partial Dagger code for this computation is shown in Figure 4.11. The Jacobi iteration loop can be expressed by only one when-block in addition to the one for initialization. The figure lists only the when-block code which implements the loop. When a processor receives the initial four boundaries (and they are expected after the initialization), it updates its grid points. After the update is completed, it sends its newly computed boundary values to its neighbors to be used in the next iteration. Then it expects new boundary values of its neighbors for the next iteration. Although this procedure is derived with exactly the same reasoning of the red-black example, it will work correctly only under some conditions. It implicitly assumes that a message received at an entry-point is the message meant in the `expect` statement. The next section presents what might go wrong in the Jacobi program.

```

dag BrancOffice red_black {
  entry init : (message INIT *msg);
  entry red_north : (message BOUNDARY *rn);
  entry red_south : (message BOUNDARY *rs);
  entry red_west : (message BOUNDARY *rw);
  entry red_east : (message BOUNDARY *re);
  entry black_north : (message BOUNDARY *bn);
  entry black_south : (message BOUNDARY *bs);
  entry black_east : (message BOUNDARY *be);
  entry black_west : (message BOUNDARY *bw);

  when init : {
    initialize();
    send_red_boundaries();
    expect(red_north);
    expect(red_south);
    expect(red_west);
    expect(red_east);
  }
  when red_north,red_west : {
    update_quarter11(rn,rw);
    send_black_boundaries();
    expect(black_north);
    expect(black_west);
  }
  when red_south,red_east : {
    update_quarter22(rs,re);
    send_black_boundaries();
    expect(black_south);
    expect(black_east);
  }
  when black_south,black_west : {
    update_quarter21(bs,bw);
    if (iteration_count < ITERATION_LIMIT) {
      send_red_boundaries();
      expect(red_south);
      expect(red_west);
    }
  }
  when black_north,black_east : {
    update_quarter12(bn,be);
    if (iteration_count < ITERATION_LIMIT) {
      send_red_boundaries();
      expect(red_north);
      expect(red_east);
    }
  }
}

```

Figure 4.9: Dag-chare for Gauss-Seidel red-black relaxation.

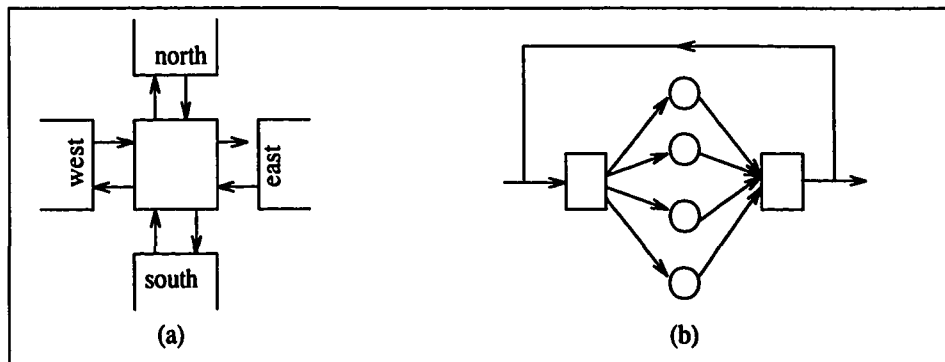


Figure 4.10: Jacobi (a) partitioning (b) dependences on one processor.

```

when north,south,east,west : {
  update(n,s,e,w);
  iteration_count = iteration_count + 1;
  if (iteration_count < ITERATION_LIMIT)
    for(neighbor=NORTH; neighbor<=WEST; neighbor++) {
      m = create_message(neighbor);
      send_message(m);
      expect(entry_name[neighbor]);
    }
}

```

Figure 4.11: Partial dag-chare for Jacobi relaxation.

4.2.6.3 Problem with the Jacobi

It is possible that messages belonging to different iterations may arrive out of order due to network delays and other factors. For instance, when a processor is expecting a message for iteration 1, it may receive one belonging to iteration 2. Such a scenario, where the computation goes wrong for the Jacobi method, is illustrated in Figure 4.12. Processors *i* and *j* are exchanging messages and doing some local computation. The message sent by processor *i* in the second iteration is delayed. When processor *i* receives a message from *j* in the third iteration, it performs the local computation and sends the message belonging to the fourth iteration. Processor *j* receives the message which belongs to the fourth iteration before the one belonging to the third iteration.

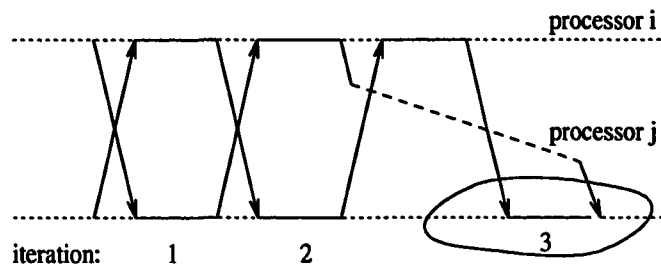


Figure 4.12: Out of order messages.

The basic difference between the red-black and Jacobi loops is depicted in Figure 4.13. The red-black algorithm performs one additional synchronization inside the loop by exchanging the boundaries a second time. This guarantees that the loop entry receives messages belonging to the same iteration (i.e., in order to send a red message, black messages must be exchanged, therefore a processor cannot send consecutively two sets of red messages belonging to different iterations).

However, in Jacobi, the loop contains only one when-block, and the synchronization is at the beginning of the loop, therefore this anomaly (overlapping with the next iteration) occurs. Now we will extend the language to control this case. On the other hand, in some algorithms, different iterations might be independent. We will show how this extension is useful to overlap such independent iterations deliberately in a controlled manner. The addition is the *reference numbers* which will be discussed next.

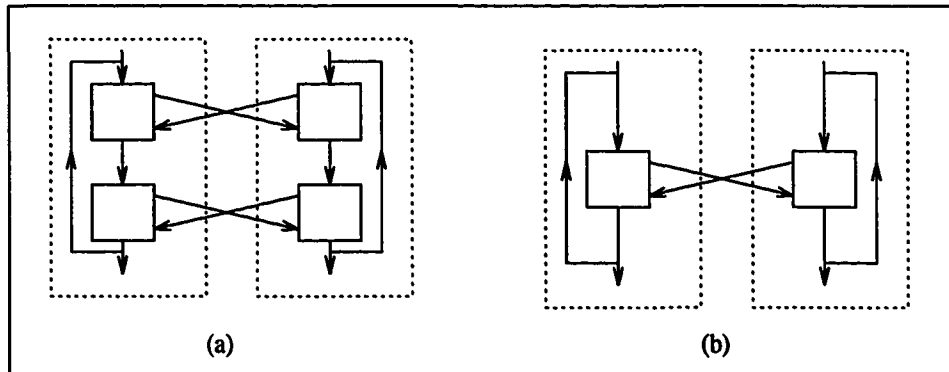


Figure 4.13: Loop structure in (a) red-black (b) Jacobi.

4.2.7 Reference numbers

The Jacobi scenario suggests that messages belonging to different phases of computations must be distinguished. To accomplish this, the Dagger language is modified to include *reference numbers*. Each message may have a reference number. Messages that belong to the same phase of computations are given the same reference number by the user. Then Dagger matches the messages with the same reference number to activate a when-block (condition variables may have reference numbers too). In other words, an instance of a when-block is scheduled for execution only if the dependence list is satisfied with the availability of messages and condition variables with matching reference numbers. The entry declaration specifies if messages for an entry-point have reference numbers with the keyword **MATCH** as follows:

```
entry entry_name MATCH : (message msg_type *msg)
```

The **expect** and **ready** statements are modified to support reference numbers as follows:

```
expect(entry_name,reference_number)
ready(cond_var_name,reference_number)
```

With this extension, a when-block can be executed many times, each with a different set of messages that have the same reference number. The reference number of messages are accessed and set by the function calls provided by the system: **GetRefNumber(msg)** and **SetRefNumber(msg)**.

```

when north,south,east,west : {
    update(n,s,e,w);
    iteration_count = iteration_count + 1;
    if (iteration_count < ITERATION_LIMIT)
        for(neighbor=NORTH; neighbor<=WEST; neighbor++) {
            m = create_message(neighbor);
            SetRefNumber(m,iteration_count);
            send_message(m);
            expect(entry_name[neighbor],iteration_count);
        }
}

```

Figure 4.14: Correct Jacobi relaxation with reference numbers.

4.2.7.1 Loop example - Jacobi loop with reference numbers

The Jacobi example is rewritten with the reference numbers in Figure 4.14. The value of the `iteration_count` is used as reference numbers for messages belonging to that iteration. After updating the points at iteration i , the new boundary values are sent with the reference number $i+1$, and the boundaries from the neighbors with reference number $i+1$ is expected. Therefore, only the when-block instance for $i+1$ will be executed with the correct set of messages. Other messages will be buffered until an expect is issued for them. In the Jacobi case, the overlapping of consecutive iterations is eliminated on purpose. However, some other computations allow overlapping of independent iterations. The usage of reference numbers in the pipelined loops will be discussed in the next section.

4.2.8 Pipelining independent iterations of a loop

Dependences in many numerical algorithms allow us to overlap execution of some iterations of a loop (for example, asynchronous iterative algorithms). Reference numbers can be used to accomplish this overlap. A generic example is to overlap the next s iterations in a loop. Messages belonging to a particular iteration have the iteration number as their reference number as before. Figure 4.15 lists the code for this generic case. Initially, the first s iterations are activated in the startup phase (`when init`) by issuing expect for these s iterations. Now, there

```

when init: {
    for(neighbor=NORTH; neighbor<=WEST; neighbor++)
        for(i=1; i<=pipeline_size; i++)
            expect(entry_name[neighbor],i+initial_iteration);
}

when north,south,east,west : {
    update(n,s,e,w);
    iteration_count = iteration_count + 1;
    if (iteration_count < ITERATION_LIMIT)
        for(neighbor=NORTH; neighbor<=WEST; neighbor++) {
            m = create_message(neighbor);
            SetRefNumber(m,iteration_count+pipeline_size);
            send_message(m);
            expect(entry_name[neighbor],iteration_count+pipeline_size);
        }
}

```

Figure 4.15: Pipelining loop iterations.

are s concurrent iterations that can be executed whenever their messages arrive. If iteration i is executing, it will activate iteration $i + s$. Therefore, there will be always a window of s concurrent iterations.

4.2.9 Multiple message entry points

Another extension to Dagger deals with entry points that receive multiple messages. This situation arises when the number of messages on which a when-block depends is known only at runtime or differs from processor to processor. An example of this is the reduction operation. In a reduction operation, some data values from every processor are combined, and the result of the reduction operation is distributed to all or some of the processors (for example, a global sum operation). An efficient and scalable way of implementing the reduction operation is to utilize a spanning tree of processors. Each processor in the tree collects and reduces the data from its children and passes the partial result to its parent. When the root receives partial results from all of its children, the final result is broadcast.

A simple reduction example (only the combining through a spanning tree) will be presented to illustrate the usage of multiple entry points. The reduction computation can be expressed in the basic Dagger language by a `when` statement which is guarded by the entry points, one for each child. However, the number of children is not fixed for all processors. In addition, as the number of processors changes, the spanning tree changes. To solve this problem, the language is extended by entry-points that can receive multiple number of messages. In Figure 4.16, a dag-chare for this example is shown. The entry declaration:

```
entry collect[n] : (message MSG *subresult[]);
```

associates a variable, `n`, with the entry `collect`. This variable is initialized to a user-specified value at the beginning (in `init entry`). The entry point `collect` now expects `n` messages stored in an array of message pointers called `subresult` in order to trigger a `when`-block. In the example, `n` is assigned to the number of children, and the `when`-block “`when collect ..`” is activated only after receiving `n` messages at the entry point `collect`, and an `expect` for the entry `collect` has been executed. It is possible to access each of these messages (if an `expect` has been executed already) individually immediately after it is received, by using the keyword `ANY` in a `when`-block as follows:

```
when collect[ANY] : { ... }
```

This example was kept simple to introduce the concept of multiple message entry points. The branch office chare `reduction` performs one reduction as soon as it is created, and then stops. In a practical parallel programming environment, on the other hand, at many places in the parallel program a reduction operation is necessary. What is needed is a reduction module which can be called by others multiple times and possibly concurrently. In order to achieve such an operation, the reduction module must be written in such a way that it handles concurrent operations. Furthermore, it must return the result of the completed reductions to the corresponding caller. The first feature is solved simply using reference numbers. The second one which requires an invocation mechanism (or interface) for concurrent operations will be discussed in the message-driven libraries section (Section 4.3).

```

dag BranchOffice reduction {
entry init : (message MSG *initmsg);
entry collect[n] : (message MSG *subresult[]);

    when init : {
        n = McNumSpanTreeChildren(McMyPeNum());
        if (i_am_leaf)
            send(to_my_parent,my_partial_result);
        else
            expect(collect);
    }

    when collect[ANY] : { combine(subresult,my_partial_result); }

    when collect : {
        if (i_am_not_root) send(to_my_parent,my_partial_result);
    }
}

```

Figure 4.16: A reduction dag-chare illustrating multiple message entries.

4.2.10 Translation and runtime of Dagger

Dagger has been implemented on top of the Charm system. The implementation has two parts: translation of a dag-chare and runtime management system. A Dagger program is a Charm program augmented with dag-chares. The dag-chares are transformed into regular chaes by the Dagger translator. An overview of the translation process from dag-chare to chare is depicted in Figure 4.17 with a simple example. Entry declarations in the dag-chare are converted into *entry-points*. The code-blocks for these entry-points are produced by Dagger translator, and they will be discussed later. When-blocks are converted into private functions. In order to perform the required synchronization and buffering operations, a control data structure (manipulated by the Dagger runtime system) is declared in the local data area of the translated chare, as well as the user-defined data structures. In addition, the translator produces some other private functions: `expect()`, `ready()`, and `when-switch()` to handle invocation of when-blocks. The following sections will discuss the important parts of these translation steps and the runtime system of Dagger.

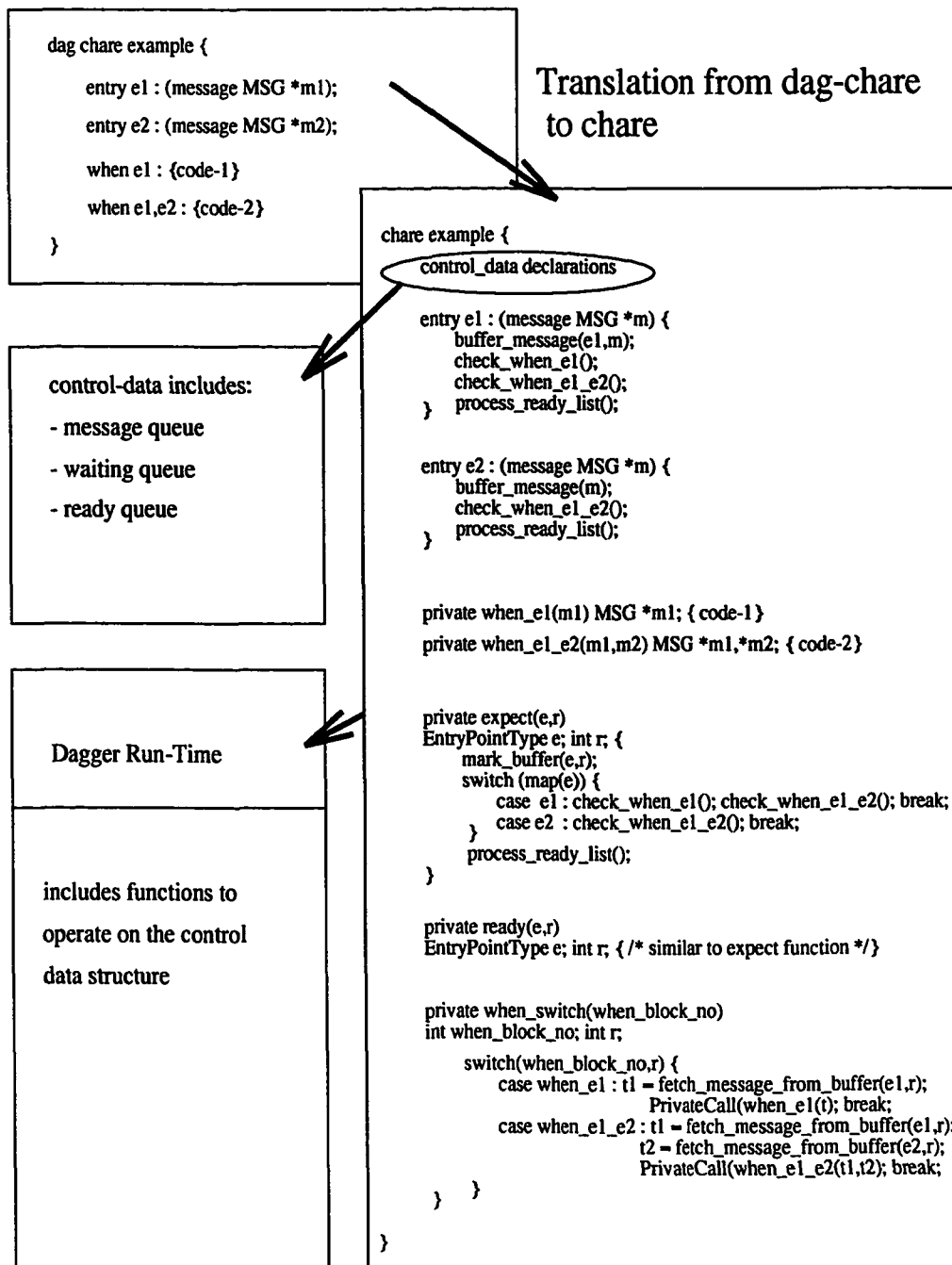


Figure 4.17: Translation of a dag-chare.

4.2.10.1 Data structures

One of the reasons for developing Dagger was to free the programmer from the tedious job of handling synchronization and out-of-order message arrivals. A dag-chare, in order to handle these problems, maintains a complex data structure. The most important part of this data structure is the message and when-block queues. These queues are not shared; each dag-chare has its own private set of these queues. A dag-chare instance maintains three queues:

- message queue,
- waiting queue,
- ready queue,

The message queue holds messages that are received but not consumed yet. It contains message buffers, which have a flag for `expect` and a pointer to the message (in addition to other fields). When a message is received by the dag-chare, it is placed in the message queue. If the message is expected, it becomes available to when-blocks. Messages stay in the queue until all the when-blocks that need the messages are completed.

The waiting queue holds the instances of when-blocks that are waiting for synchronization. These when-block instances are in the queue because their dependency list is partially satisfied (they are waiting for the rest of messages and/or condition variables). Each element of the waiting queue has a counter and a list of message pointers. This counter is initialized to the number of entry-points and condition-variables in the dependence list. The counter is decremented whenever one of the condition-variables in the dependency list is set or a message becomes available for an entry in the list.

The third queue is a list of when-block instances that are ready to execute. A when-block instance is transferred from the waiting queue to the ready queue when its dependence list is completely satisfied (*i.e.*, synchronization counter is zero). The Dagger runtime consumes this queue by executing the when-block instances.

For all of these queues, the most common operation is a search with a key “entry-point and reference number” (or when-block and reference number). The information about entry-points and when-blocks (number of them, their range, etc.) is known at compile time. The search for this part of the key is done by a direct array access. However, reference numbers are dynamic

entities (*i.e.*, their range cannot be predicted statically). Therefore, this access requires a search operation.

4.2.10.2 Entry-point functions

As shown in Figure 4.17, each entry declaration is converted into an entry-point with a code-block. This code-block (produced by the Dagger translator) first enqueues the message. If the message is already expected, the code-block checks the when-blocks that depends on this message. The list of dependent when-blocks is known at compile time, therefore the Dagger translator produces the proper code that checks those when-blocks only. The when-block instances are searched in the waiting queue first. If no instance of a dependent when-block is found, then an instance is created and put into the waiting queue.

4.2.10.3 Expect and Ready functions

The translator produces separate `expect()` and `ready()` functions for each dag-chare in the Dagger program. These functions perform work similar to the entry functions.

4.2.10.4 When-switch function

Remember that when-block definitions are converted into functions, these functions are called from the `when-switch` function, which is produced by the Dagger language for each dag-chare. `when-switch` prepares messages as input to the when-block functions and then invokes the functions.

4.2.10.5 Process-ready-list function

This function executes when-block instances that are in the ready-list. When there are no more instances in the list, the control is returned to the Dagger runtime system. It is called from entry points.

4.2.10.6 Miscellaneous functions

In addition to these basic implementation details, there are a few more points specific to Dagger. The `DagExit()` function is provided to exit a dag-chare instead of the return statement of C.

This call was necessary in order to keep the consistency of the runtime data structure of Dagger. The runtime system automatically frees messages also. It keeps track of when-block executions. When it detects that all the dependent when-blocks of a particular message have been executed, the memory hold by the message is freed. This option can be cancelled if the user prefers to reuse messages. However, the user must do exactly the same bookkeeping as Dagger does in order to free messages.

4.3 Message-driven libraries

Libraries constitute an important part of software development process. They provide reusable, portable code, and they hide details from application programmers. There are many SPMD parallel libraries for commonly used kernel operations, such as numerical solvers, fft, etc. As discussed in Chapter 2, SPMD style does not encourage usage of multiple concurrent libraries. On the other hand, message-driven style encourages creation of smaller and more reusable modules. Therefore, we expect libraries to be a major strength of message-driven systems in the future.

However, developing message-driven libraries requires a different set of assumptions. The organization of the code and the nature of the interfaces across modules is different. The issues involved in message-driven inter-module interfaces must therefore be understood before one can undertake development of large sets of message-driven libraries. The main reason why message-driven libraries have to be treated differently from SPMD libraries is the split-phase nature of inter-module exchanges. When a module calls another module, it must simply deposit the data it wishes to deposit, and then continue or suspend. Eventually, the other module will return data and control to the calling module in the form of a message.

4.3.1 Problems with libraries in SPMD style

In traditional SPMD model, library computations are invoked by regular function calls. The library call blocks the caller. After the library computation has been completed, the library module returns the result and control to the calling modules. There are major performance disadvantages of libraries in SPMD style. These can be listed as:

1. Idle times in the library computation cannot be utilized if there are other independent computations (as discussed in Chapter 2).
2. Caller modules must invoke the library even though only a subset of the processors are involved in the library computation from the caller's point of view.
3. Library computations must be called in the same sequence on each processor.

These drawbacks force SPMD style programmers to use flat codes (*i.e.*, merging potential library modules into the main or caller module) for better performance. The second requirement, namely “every processor has to call”, might be a real bottleneck for developing large efficient programs. Consider the following scenario. Assume that only one processor initiates and needs the result of a library computation. The computation is a parallel one and involves a set of processors. Figure 4.18 depicts the interaction of an SPMD and a message-driven program to such a library. In this particular example, the message-driven program (part (a)) calls the library if it is the processor that initiates and needs the result of the library computation. Note that the call is nonblocking. Therefore, the calling processor can potentially continue to execute its next computation block while the library computation is going on concurrently. The library itself may involve computations on an arbitrary subset (or all) of the processors. In the message-driven program, the caller module does not have to call the library module on all the processors. The library itself will send messages to its components only in those processors involved. The SPMD program, on the other hand, has to call the library on all processors. When the library returns the control, the program checks if it really needed to call the library. If that is the case, then it performs post-library computation.

Finally, library calls must be made in the same sequence on every processor. For example, assume two reduction operations, *r1* and *r2* needed to be computed. If one processor first calls *r1* and the other calls *r2*, then neither of them can succeed.

4.3.2 Message-driven execution and library interface techniques

The interface between message-driven programs and message-driven libraries are different from the interface for SPMD style. Since computations are split-phase in message-driven style, library calls must provide a return address for the result or completion. The three steps of a simple library interaction in message-driven style are: a) creation of the library object, b) invocation

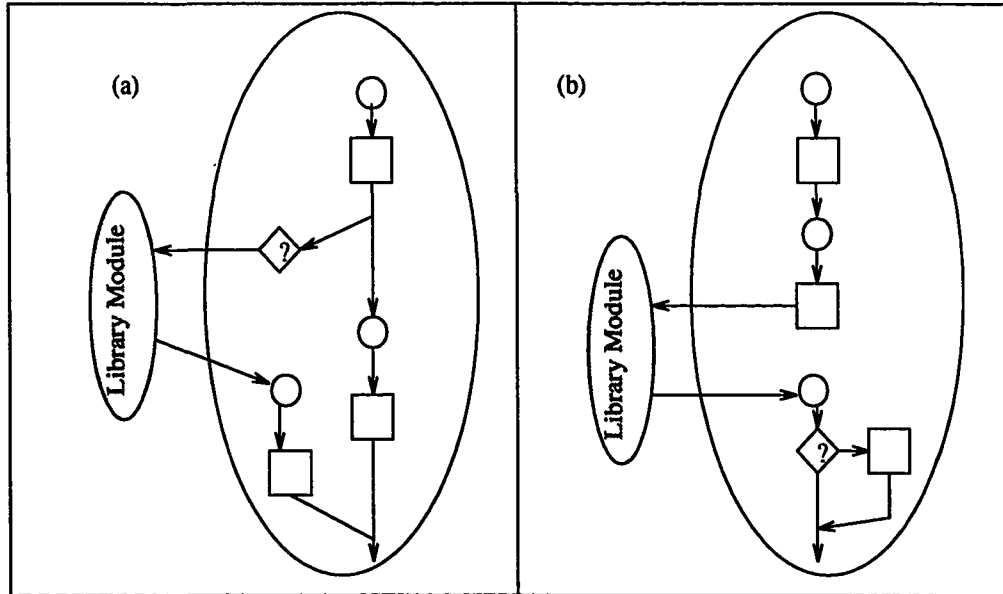


Figure 4.18: Library call.

of a library computation, c) reception of the result. These points will be discussed within the Charm environment and a library calling scheme will be defined.

The first step in a typical Charm program is to create instances of library modules. Once the instance is created, it can be invoked by all the participating computations using its unique instance identifier. This requires the participating computations (or chares) to have the library's instance identifier. The creation phase, thus, consists of creating the instances of the library and distributing the instance identifier to users of the library (clients). The creation of a library module can be encapsulated into a function. The library exports this function, `create`, that handles all the steps of a creating the library (the user does not have to know the details of the library). There are two ways to invoke the create function:

```
lib_instance = LIB::create()
```

or

```
LIB::create(chareid,entrypoint)
```

The first form can be used only at the beginning of the program (*i.e.*, in the `CharmInit` entry point), while the second one is a nonblocking `create()` function, which returns the instance number to the address (`chareid,entrypoint`) in a message.

A library can be invoked by sending a message to it. The library module exports definitions of types that clients need, such as messages and names of chares and entry points in an interface file (see Section 2.5.1.4). The message has to be allocated, filled, and sent to the library module. All these steps can be encapsulated into a function that the library exports. This function needs input data from the caller (optional) and a return address. The result can be returned in a message. In this case, the library invocation call is as follows:

```
LIB::request(lib_instance,data,my_chare_instance,entry_point)
```

The library instance `lib_instance` is invoked, and the result will be returned in a message to `entry_point` of the caller chare. A second option is to receive the result by function call

```
LIB::request(lib_instance,data,result.buffer,function_ptr)
```

This call provides the library with input data, a pointer to the result area, `result.buffer`, where the library directly puts the result there, and `function_ptr` which is a public function that the library module calls when the result is ready.

4.3.2.1 Concurrent library calls

In the previous section, the interface for creating and invoking message-driven library computations is described. If the same library module needs to be called multiple times and concurrently, what should the interface be? There are two options; the first one is to create multiple instances of the library module as shown below:

```
id1 = LIB::create();
```

```
id2 = LIB::create();
```

and different instances can be invoked concurrently as follows:

```
LIB::request(id1,data1,mychareid,e1);
```

```
LIB::request(id2,data2,mychareid,e2);
```

The second option is to design the library module such that it handles concurrent calls (as described in Section 4.2.7). The caller provides a reference number. The library maintains a separate environment for each reference number to service the requests concurrently. A typical usage of this scheme is to create an instance of the library and invoke the same instance with different reference numbers for each distinct request.

```
id1 = LIB::create();
```

```
LIB::request(id1,data1,1,REQUEST_1,mychareid,e1);
```

```
LIB::request(id1,data2,1,REQUEST_2,mychareid,e1);
```

The reference number of the result message is set to the REQUEST_1 or REQUEST_2. If the return is by a function call, then the reference number is passed as an additional parameter to the return function.

4.3.2.2 An example of message-driven library usage

Figure 4.19 lists a sample code showing how to use a library. The library (a global sum operation) exports the functions for creation and depositing data to be summed. The main chore creates two instances of it (in order to show concurrent usage of library instances). Then it makes the instances known to everybody (ReadOnly variable is a type of abstract data structure which is supported by Charm for this purpose). Every node of redn_client deposits its data by calling the deposit function. The library provides two deposit calls. Deposit_msg supports return-by-message mechanism. Deposit_f supports return-by-call. Note that, Deposit_f takes one additional parameter, z, a pointer to store the result. The library stores the result at this area and calls the return-function. Then the function accesses the result.

```

interface module Sum {
    message {ChareNumType id; varSize int data[]; } MSG;
    Create();
    Deposit_f();
    Deposit_msg();
}

-----

#include "sum.int"
module redn_example {
#define VSIZE 10
readonly ChareNumType redn1,redn2;

chare main {
    entry CharmInit {
        redn1 = Sum::Create(VSIZE);
        redn2 = Sum::Create(VSIZE);
        client = CreateBoc(redn_client,redn_client@start,msg);
        ReadInit(redn1);
        ReadInit(redn2);
    } ...
}

BranchOffice redn_client {
int x[VSIZE],y[VSIZE],z[VSIZE];
ChareNumType mybocnum;
ChareIDType mybranchid;
    entry start :(message MSG *msg) {
        mybocnum = MyBocNum();
        MyBranchID(&mycid);
        Sum::Deposit_f(ReadValue(redn1),x,recv_result,&mybocnum);
        Sum::Deposit_msg(ReadValue(redn2),y,z,recv_entry,&mybranchid);
    }
    entry recv_entry : (message Sum::MSG *msg) {
        use(msg->data);
    }
    public recv_result(id)
    ChareNumType id;
    { use(z) }
}
}

```

Figure 4.19: Using the reduction library.

4.4 Related work

The Dagger notation was developed to specify the partial order between subcomputations (or methods) within an object or process. The syntactic mechanism used for this purpose included when-blocks, which are sequential blocks of computations guarded by conditions. To put Dagger in context, we now compare it with past work on the use of guards and other methods for coordinating subcomputations within an object.

Guarded Commands

The idea of associating guards with computations, as done for when-blocks, is related to the notion of guarded commands [27]. Guarded commands are sequential statements where each statement is prefixed by a boolean expression. The statement is said to be open if its guard is true. The nondeterminism in a sequential code is controlled by two guarded constructs: the alternative construct and the repetitive construct. An alternative construct is of the form:

```

if guard1 → statement1
[] guard2 → statement2
[] ...
[] guardn → statementn
fi

```

The alternative construct is executed by selecting any one of the open guarded statements. If none of the statements are open, then an error occurs. The repetitive construct is similar to the alternative one, i.e., it selects the open statements arbitrarily, but it repetitively evaluates guards and executes them until all the guards are false. If none of the guards are true, the command is skipped (or an error occurs). Guarded constructs are a generalization of non-deterministic if statements. They do not express concurrent subcomputations. On the other hand, Dagger is a language to express concurrent computations. Furthermore, when-blocks in Dagger are executed whenever their conditions are satisfied, i.e., a when-block is delayed until its condition is satisfied, whereas guarded commands are skipped if the condition is not true at the time the control reaches them.

In the context of concurrent programs, guard-like notations have been used to specify various synchronization properties. Such use of these notations can be found in studies including

Communicating Sequential Processors [54], Distributed Processes [50], Path Expressions [13], and Ada.

Communicating Sequential Processes (CSP)

CSP, described by Hoare, incorporates the guarded commands and Dijkstra's parbegin constructs [26] for concurrent programming. Processes communicate through synchronous messages. During the communication both sender and receiver processes block. The guard expressions can include the readiness of a sender or receiver process. An input guard is satisfied when the sender process is ready to communicate its output. With this extension, a process can wait for multiple inputs by using the alternative guard construct. When a sender process in the alternative list is ready, then the receiver synchronizes with that sender. After the communication is completed, the corresponding statement is executed in the guarded construct.

The guarded inputs together with the alternative construct have similarities to the when-block construct of Dagger. However, it is difficult in CSP to achieve the adaptive overlap of subcomputations of a process as in Dagger due to the blocking nature of the communication scheme between processes.

Distributed Processes

Distributed Processes was proposed by Hansen to support concurrent real time programs. A Distributed Processes program contains a fixed number of concurrent processes. Processes can call procedures defined within other processes. The synchronization among processes is achieved through guarded commands and guarded regions. Guarded regions are similar to guarded commands, but they are not skipped if the guard expressions are false. They suspend the process until their conditions become true. The state of a suspended process can change when another process invokes a procedure in the suspended process. If this invocation causes the guard of a suspended region to be true, then the process resumes.

The when-block construct of Dagger is closely related to the guarded regions. There are two types of guarded regions that one can use in a process, `when` and `cycle` regions:

```
when  $B_1:S_1 \mid B_2:S_2 \mid \dots \mid B_n:S_n$  end
cycle  $B_1:S_1 \mid B_2:S_2 \mid \dots \mid B_n:S_n$  end
```

where B_i are guard expressions and S_i are statements. The `when` guarded region waits until at least one of the guards becomes true. If more than one guard is true, then one of them is

```

cycle
  B1:
    S11
    cycle B12 : S12 | B13 : S13 end
    S14
  B2: S2
end

```

Figure 4.20: Distributed Processes example.

selected arbitrarily. The `cycle` region is the repetitive version of `when`. `Cycle` completes its execution when all the guarded statements in the cycle are processed.

Major differences between Dagger and Distributed processes (within the context of ability to overlap adaptively) that makes Dagger advantageous are (a) in Distributed Processes, the caller process blocks until its request is accepted and completed, (b) If the guarded regions are nested, then the process can not switch between regions.

A blocking caller (a) prevents a process from issuing multiple concurrent requests. However, in Dagger, many programs benefit from being able to issue multiple requests as shown in Chapter 6. The nested regions construct decreases the opportunities for adaptive selection (nondeterministic selection) of requests as illustrated by the following example. Consider the Distributed Processes code in Figure 4.20. If B1 becomes true before B2, then the sequence of statements, which includes another guarded region, is executed until completion. It might happen that, while the process is waiting idly for some guard in the inner cycle, B2 might become true and the process cannot switch to it.

However, cycle regions can be flattened to overcome this problem. But, it requires introducing additional flags and counters in the guard expression to express the partial order in the nested structure. Finally, implementation of guarded regions is more expensive. The guards have to be evaluated whenever the state of a process changes. The state of the process can change whenever an assignment involves variables in the guard expressions. In Dagger, only message arrival and `expect/ready` statements trigger such evaluations, and it is easier to produce optimized code.

Path Expressions

Path expressions provided by Campell and Habermann was developed to ease the expression of synchronization between concurrent processes. The explicit programming of such synchronization requirements with P and V operations complicates the program. The purpose of path expressions is to provide the user with a better way to express such synchronization. The mechanism specifies how a procedure (subcomputation) is allowed to execute in relation to others. Processes, which are executing concurrently, are allowed to call the procedures according to a specified order given in the path expressions. A path expression has two synchronization actions: sequence and selection actions. The sequencing specifies that the procedures are called in a sequence. For example, the path expression

`path p;q end`

specifies that procedures q cannot be called before p is completed. Any process that attempts to call q is blocked until another process calls p. The selection action permits only one of the procedures to be called from a set of them. The path expression

`path p,q end`

specifies that only one of p and q can be called. These two forms of synchronization can be combined in one expression.

Path expressions are not adequate to express partial orders in the way when-blocks express them. Consider the following example. The dependences among the procedures p, q, r, and s are such that q and r must follow execution of p, s can be executed after q and r are completed, and q and r can be invoked in any order. When-block constructs allows us to choose q and r in any order, whichever is requested first, after p is completed. However, in path expressions, we have to enumerate all possible total orderings in order to represent the partial order. In other words, we can write the path expressions as:

`path p;(q;r,r;q); s end`

```

select
  when condition1  $\Rightarrow$  accept entry1
  or when condition2  $\Rightarrow$  accept entry2
  or ...
  else statement
end select

```

Figure 4.21: Ada select statement.

This is not practical since number of enumerations is exponential in the number of procedures. Finally the caller process blocks which further limits the concurrency.

Ada

In Ada, synchronization amongst concurrent processes is achieved with the *rendezvous* mechanism. This mechanism of Ada is heavily influenced by CSP and Distributed Processes. Two processes, a caller and a server, synchronize when the caller calls a particular entry in the server, and the server issues an *accept* statement to receive the call. The nondeterministic choice of accepting requests comes from the *select* statement. Figure 4.21 shows the select statement. When the server reaches the select statement, it evaluates the conditions (guards). If one or more conditions are true, then one of them is selected arbitrarily, and the rendezvous occurs with the corresponding caller.

Dagger is more powerful, within the context of adaptive selection of requests, because, in Ada, the caller process blocks until the server accepts and completes the request. In addition, the server may also block. This happens when one of the conditions is true in the select statement and the corresponding caller process is not ready for the rendezvous.

HeNCE

Partial orders are also used to express task graphs. HeNCE [8] (Heterogeneous Network Computing Environment) is an integrated graphical environment for creating and running parallel programs over a heterogeneous collection of computers. It allows one to define dependences among subroutines and a mapping of subroutines to processors. A task starts after all of its predecessors have completed, and it returns its output to its successor tasks. Task graphs are

only marginally related to Dagger because Dagger deals with coordination of subcomputations within an object, and the subcomputations are on the same processor.

Message/Data Driven Models

More pertinently, the problem of local synchronization within a parallel object (process) has been dealt with in other research including that on Actors, ABCL, TAM and macro dataflow.

The work on enable sets was done in the context of a broad problem that arises in concurrent object oriented systems. This is the problem of how to inherit synchronization constraints. When a subclass is created from an existing class, the individual methods are inherited as well as synchronization constraints. Inheriting the synchronization constraints is not as easy. This fundamental problem has been addressed by many other object oriented systems. A system such as Dagger can be thought of as specifying synchronization constraints on the object to decide in what sequence different methods can execute.

The original Actor model as described in [1] is purely message driven. The issue of synchronization within an actor was addressed in [91] which proposed the *enable set* construct. Using this, one may specify which messages may be processed in the new state. Any other messages that are received by an actor are buffered until the current enable set includes them. Thus, this construct is analogous to the expect statement in Dagger. However, there is no analogue of the when-block, viz. a computation block, that can be executed only when a specific group of messages have arrived. A more recent paper [37] supports a much more complex model which subsumes synchronization of multiple actors depending on message sets. It should be noted that Dagger/Charm provides a programming model that differs from Actors in many ways. The discussion above focuses only on how they deal with message driven execution.

Other languages, such as CC++ [15], also address to some extent the issue of synchronization within an object. In CC++, the par construct can be used to express partial orders. However, CC++ is a thread based model and the actions in alternate threads (or par constructs) can interleave at any granularity. In contrast, in Dagger a when-block, once it begins execution, will finish its execution before control is transferred to another when-block in the same object. Although there may be advantages of finer granularity of interleaving in a thread based model, we believe that the Dagger method leads to cleaner code which reduces possibilities for bugs because of interleaving. Secondly, Dagger does not depend on the availability of a threads package on the underlying machine, and neither is it affected by inefficiencies of such a package.

More recent work on ABCL, a concurrent object oriented system, includes synchronization schemes. A object uses synchronizer constructs to express dependences between the object and the set of messages that it can process. A synchronizer is a combination of guard expressions and method sets (subcomputations in the object). The guarded expressions allow one to express dependences involving multiple messages. Therefore, they are similar to when-blocks of Dagger.

The TAM compiler [30] built on Active Messages has some similarities to Dagger. In TAM, as messages always enable the corresponding threads of an activation frame, there appears to be no way of buffering unexpected messages. Counters and flags for synchronizing on arrival of multiple messages are explicitly maintained. However, TAM is meant as the back end for a dataflow compiler as opposed to a language meant for the application programmer. So these inconveniences may not be of much consequence.

Macro dataflow [33, 41, 70] approaches share with us the objective of message driven execution and local synchronization. Initially dataflow computations were defined with static dataflow graphs. Dagger programs without reference numbers corresponds to this model. Later, multiple instances of loop iterations and functions are handled in the tagged token model. The Dagger programs with reference numbers corresponds to this model. Dagger can be thought as a software implementation of tagged dataflow model on message passing machines. Much of the past dataflow work in this area was aimed at special purpose hardware. Also, these approaches were often meant to be used as a back-end for compilers. Thus the inconvenience of maintaining counters and buffers explicitly is not considered significant. On the other hand, Dagger is meant to be used at the programming language level. Dagger can be used to build higher level systems also. Our experience with using Dagger as back-end for a compiler for a data parallel language [66] indicates that Dagger might provide a more convenient intermediate language than macro dataflow.

Chapter 5

Simulating message-driven programs

In this chapter, a methodology for simulating message-driven programs is presented. The information that is necessary to carry out such simulations is identified, and a method for extracting such information from program executions is described. It takes advantage of Dagger for simulations. As we will be describing, Dagger notation developed in Chapter 4 plays an important role in the methodology for simulation. The simulation developed here is used for studies reported in Chapter 6.

5.1 Introduction

An accurate performance prediction of parallel computations plays an important role in designing parallel algorithms and machines. Many computational complexity models have been derived for parallel algorithms. Although these analytical approaches are very useful to determine the fundamental performance limits of parallel algorithms, they are often inadequate to analyze parallel computations and the interactions between computations and parallel architecture. Particularly, the load unbalance, scheduling, synchronization, and the dynamic properties of parallel computations make analytical approaches too difficult, if not impossible, for this purpose.

Simulation techniques offer a more realistic analysis in this regard, and they have been used extensively to predict the performance of computers. There are various simulation techniques

which are suitable for different purposes depending on the accuracy of the prediction desired and the complexity of the simulator itself. For example, one can emulate the user code instruction by instruction in the simulated environment. Although this method provides very accurate results, the simulator is itself very costly in terms of both development and computation (for instance, emulating every instruction of a parallel code on a 1000 processors). Another approach, which makes simulations more affordable, is to use an abstract model of the computation instead of emulating a specific computation. The model may contain some statistical properties such as average number of messages sent, average computation size, etc. As no user computation is executed during the simulation, the simulation time (and the computation power required) is small. However, this approach too may not capture enough details of the computation and is not useful to predict the performance of a specific program. Trace-driven simulation is another approach which combines the advantages of both. Its complexity is in between and yet it is powerful enough to capture the details of the computation. In a trace-driven approach, the simulator uses the traces of a computation. A trace is a timely ordered sequence of events that happened during the execution of a program. Traces are collected from an actual execution of the program, and they are fed to the simulator. The simulator, then, executes these traces on a model of the new system. Trace-driven simulation has been successfully used in studies of uniprocessor systems such as memory designs, cache performance, [86], etc. In such studies, the order of events in the system to be simulated has been assumed to be deterministic. Therefore, the changes in the simulated environment affects only the length of the time interval between the events, not the actual sequence of events.

Trace-driven simulation has been extended to study parallel computations also [23, 29, 58, 93]. However, application of trace-driven simulations to the study of parallel systems poses a problem. The behavior of a parallel computation may change under a new environment. Some of trace-driven studies of parallel computations considered SPMD style computations where the behavior of the algorithm was the same despite the changes in the environment [58]. In some other works, this problem has been addressed [23, 55]. These studies combined trace-driven simulation and real execution of user code (hybrid method: trace-driven and execution-driven) to study the performance of shared memory systems. This approach is difficult and expensive since it requires execution of a part of user code during simulation. Doing the hybrid simulation for message passing systems would be more difficult and impractical.

Message-driven execution is based on the ability to run computations in different orders. Therefore, the simulation of message-driven computations involves dealing with this difficulty. In a message-driven computation, messages may arrive in a different order due to numerous reasons in the new simulated environment. The execution trace of the program becomes invalid at that point because the rest of the computation is different from the traces. In order to achieve accurate simulation, it is necessary to reconstruct the remaining sequence of computation steps. This is impossible, in general, without rerunning or interpreting the program instruction by instruction as discussed in Chapter 3. In some special cases, using the knowledge about the algorithm/computation model, the sequence of computations can be reconstructed.

In the next sections, we will describe a method to simulate a particular class of message-driven computations accurately.

5.2 A sufficient condition for accurate simulation

In this section, we will define a special class of message-driven computation, and we will show how a trace-driven simulation of these computations can be achieved accurately. In this computation model, a parallel computation is composed of processes that respond to messages directed to them. A process has entry points at which messages are received and subcomputations that are triggered by the availability of messages. These subcomputations (which belong to the same process) are atomic in the sense that they execute sequentially until the completion.

The messages and subcomputations in a process can be visualized as a dependency graph. As an example, consider the computation, which is expressed as a directed acyclic graph (dag) G , in Figure 5.1. The vertices of G are subcomputations represented by rectangles, and message availability points are represented by circles. A subcomputation may be executed if and only if its immediate predecessor nodes are completed. If the predecessor is a computation node, then it is completed if it is executed. If the predecessor is a circle, then “completed” means that it received its message and its predecessor nodes are completed. Note that message arrival and message availability are different from each other. A message may arrive at any time, but it becomes available only after the predecessors of its corresponding circle node have been completed. For example, the computation $C1$ in Figure 5.1 can be executed after a message has arrived at $E1$ and the computation $C0$ has been finished. The whole computation is a collection

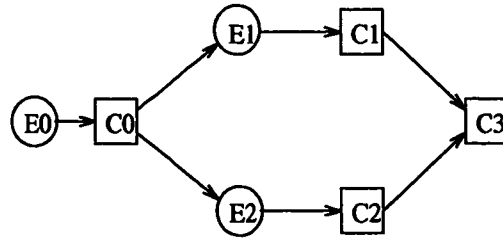


Figure 5.1: A dag computation

of such processes. Some of processes may exist on different processors that may execute in parallel, or some of them may exist on the same processor that execute in a timesharing manner.

The execution trace of a program is a timely ordered sequence of events that happened during the actual run of the program. As the runtime conditions (and communication delays) change, the execution trace of the program changes also. In order to simulate a message-driven program by trace-driven simulation, the simulator needs to reconstruct the traces under the new runtime conditions. The new sequence of traces can be reconstructed from a particular execution trace by using the dependence information. The dependences among the blocks and messages forms a partial ordering. The blocks that are incomparable¹ in this order belong to concurrent paths that may be executed in an arbitrary sequence. Depending on the order of arrival of messages, we can construct a new total order of computations that does not violate this partial order. Consider the previous example again. Let's assume that message arrival order in the real execution traces is (E0,E1,E2) and the order of subcomputations is (C0,C1,C2,C3). During simulation in a new environment, assume that messages arrive in the order (E2,E0,E1). When a message has been received at E2, the subcomputation C2 can not be executed because C0 is not completed yet. Later, when the message for E0 arrives, we can execute C0. Now the subcomputation C2 is ready for execution. However, in the execution trace, the next one is the subcomputation C1. In the partial order, C1 and C2 are incomparable, that is, they can be executed in any order. So we can execute C2. When E1 receives a message, then we can execute C1 and then C3. So we executed the blocks in the order (C0 C2 C1 C3) without violating the partial order.

¹Two blocks are incomparable if neither is a successor or predecessor of the other.

If the parallel computations are expressed in the model we just described, then

- dependences among messages and computations, and
- an instance of execution trace

are sufficient to achieve an accurate trace-driven simulation of such programs.

Dagger notation which was developed in Chapter 4 provides a way of expressing such dependences. Instead of tracing the arrival and processing of messages, one can obtain somewhat more refined traces by tracing the execution of individual when-blocks as explained in the next section. Message dependence graphs generated at compile time along with runtime traces of when-blocks are adequate to simulate message-driven execution under varying message sequences as long as the following condition is met:

- two when-blocks that are incomparable do not have data dependence between them (*i.e.*, all data used in a when-block is defined in one of the predecessors only).

This condition is usually satisfied in most application programs since violation of this condition would lead to indeterminate output.

5.3 Dagger programs and automatic trace generation

The traces that are necessary for the simulator can be produced in different ways. One way is to generate them synthetically. Another way is to implement the program and get traces from the real execution. In this section, we will describe how Dagger can help to produce the required dependence information and traces.

The dependence information is extracted from programs by the Dagger translator. This part is relatively easy because the language is designed to express such dependences.

In order to get the execution traces, the Dagger translator modifies the user program by inserting the necessary code that will produce traces. A minimal set of events are defined that contain enough information for the simulation. These events are listed below:

begin when-block : instance, when-block-id, time

send : destination(processor,instance,entry-id),length,priority,time

broadcast : destination(instance,entry-id),length,priority,time

expect : instance, entry-id

ready : instance, condition-var

set multiple message entry entry-id, count

set condition variables condition-var, count

end when-block time

Note that, the reception of a message is not traced because that information will be created by the simulator during simulation. The traces are collected in a buffer in the main memory. When the buffer becomes full, it is written into a trace file. Every processor produces its own trace file.

5.4 The parallel machine model

We have defined the computation (programming) model in the previous sections. In this section, we will define a model for the parallel machine on which these computations are to be simulated.

Despite the large diversities among the parallel machines, they have a common property: accessing remote data takes longer than accessing local data. The machine model, which will be described here, emphasizes this property. In this model, a parallel machine is a collection of complete processing nodes interconnected by a communication network as shown in Figure 5.2.

Each node consists of a processor, a local memory, and possibly a communication processor. The communication processor interfaces the processor to the network. It can access the local memory and interact with the network without blocking the processor, therefore it releases the processor from most of the communication-related tasks. The network provides communication between any two processor nodes. In reality, there exist various communication network structures with different topologies and communication protocols. From our point of view, the network provides data transfer with a latency that may depend on the network load in an arbitrary fashion, and it has a finite capacity.

Communication between two processes involves a number of steps (and delays at each of these steps). We will explain these steps by an example which is depicted in Figure 5.3. In this

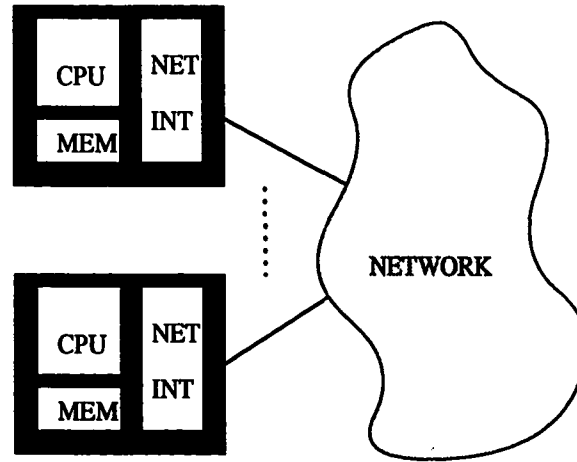


Figure 5.2: The parallel machine

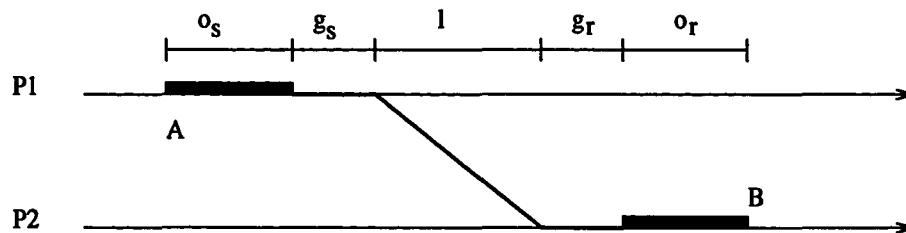


Figure 5.3: Sending a message

example, Process P_1 sends a message to process P_2 . At time A , P_1 starts sending the message. The processor spends o_s time units for the send operation. Then the communication processor interacts with the network and spends g_s time units. After l time units, the message arrives at the destination node. The receiving communication processor receives the message, and after g_r time units, the message becomes available to the processor. The receive operation takes o_r time units at the processor.

The total delay that a message experiences, or the time between the process issues a send operation, and the message becoming available for the receiving process, is the sum of these delays:

$$o_s + g_s + l + g_r + o_r$$

symbol	definition
p	number of processors
g_r	minimum time difference between handling of two messages by the receive communication co-processor
g_s	minimum time difference between handling of two messages by the send communication co-processor
o_r	the overhead incurred in receiving a message by the processor
o_s	the overhead incurred in sending a message by the processor
l	latency of the network which depends on network load.

Table 5.1: Machine parameters

The sender processor is blocked only o_s time units (similarly the receiving processor is blocked o_r time units). During the other parts of the delay, the processor is free to perform computation. Similarly, a coprocessor is blocked by g_s (or g_r) time units. This limits the amount of and the number of messages a processor can inject and receive from the network per unit of time. Therefore, a message-driven computation can tolerate the $g_s + l + g_r$ part of the remote information access delay.

Each of these parameters has a fixed part and a variable part that depends on the size of messages. As in many studies, we chose to model each of the parameters as:

$$\alpha + \beta n$$

where α is the startup cost, β is the time per data item, and n is the number of data items in the message. The network latency has been modelled by one more factor – the network load – in addition to α and β . Table 5.1 lists the communication parameters.

In addition to these parameters, the model includes capacity limitations similar to those described by [21] for a more realistic approach. The network has a finite capacity. This is modelled by blocking the sender communication processor if the volume of messages traveling in the network is above a threshold. The sender communication processor has a finite buffer to hold messages deposited by the processor also. If the sender runs out of buffer space, then the processor is blocked. These capacity constraints may cause deadlocks during the simulation (for example, when a circular dependency forms among the processors and all of them run out of

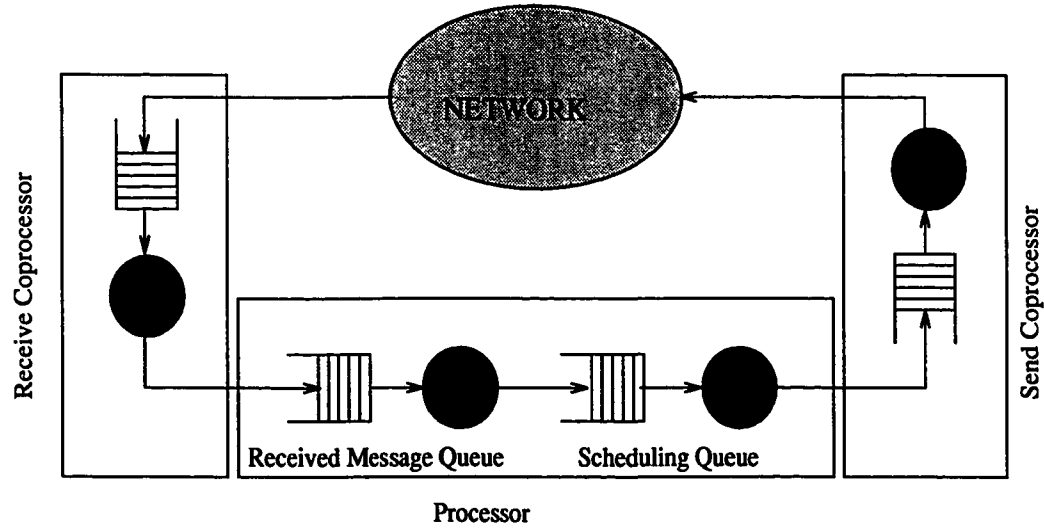


Figure 5.4: Message queues

buffers). Deadlock can happen rarely, and the simulator does not try to avoid such situations. This model subsumes the LogP model presented in [21].

Before discussing the simulator in the next section, some brief description of the message queues in the model within the context of message-driven execution will be useful. Due to the asynchronous (or nonblocking) nature of the communication, messages are queued at different states. Figure 5.4 shows the various message queues in the machine model. The coprocessors have FIFO queues for incoming and outgoing messages. The processor, on the other hand, has two message queues. One is the incoming message queue that holds the messages sent by the coprocessor, and it is managed by the native operating system, typically by a FIFO strategy. The second one is the scheduling queue, which is operated by the Charm runtime, and is not necessarily a FIFO queue (it may be a LIFO or prioritized queue).

The processor can be visualized as two processes with a producer-consumer relationship. The first process fetches messages from the receive-queue and puts it into the scheduling queue. The second one is the user process which picks the next one from the scheduling queue and processes it. The scheduling queue is necessary in order to apply different scheduling strategies within the context of message-driven execution.

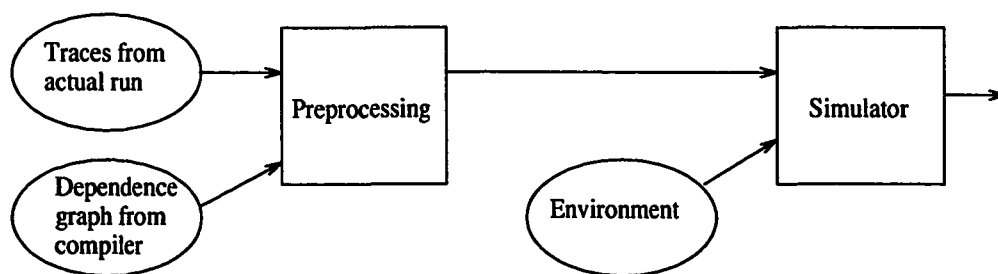


Figure 5.5: Simulation system

5.5 Simulator

In the previous sections, we have established that the dependence information and traces of the computation blocks (when-blocks) are sufficient for simulating message-driven simulation. In this section, we will discuss the design of the simulator.

The simulator consists of three major components: the preprocessor, the parallel machine simulator, and the trace interpreter. A simulation session, as shown in Figure 5.5, starts with the preprocessing of the execution traces. The output of this stage then is interpreted by the the trace interpreter on the simulated parallel machine model.

5.5.1 Preprocessor

The Charm/Dagger programming system allows us to use multiple module compilation, *i.e.*, independently compiled Dagger programs can be linked at runtime. The Dagger translator produces a separate dependence information for each module. Therefore, these individual dependence information and the runtime trace information are reconciled, and a single dependence graph and trace information are produced. In addition to this, the preprocessor converts all timing information to relative times. The traces from Dagger programs contain absolute times. For example, a when-block trace with absolute times might be as:

```

when-block instance A started at time t1
send message B at t2
when-block instance A ends at t3
  
```

BEGIN $\langle p, b, i, r \rangle \langle t \rangle$	beginning of a when-block trace which is identified by the quadruple $\langle p, b, i, r \rangle$, with elapsed time t
SEND $\langle p, b, i, r \rangle \langle l \rangle \langle prio \rangle \langle t \rangle \langle c \rangle$	Send a message to the entry $\langle p, b, i, r \rangle$ of length l with priority $\langle prio \rangle$ after t time which blocks the processor c time
BROADCAST $\langle p, b, i, r \rangle \langle l \rangle \langle t \rangle \langle c \rangle$	Same as send
EXPECT/READY $\langle b, i, r \rangle \langle c \rangle$	An expect/ready statement for $\langle b, i, r \rangle$, elapsed time for this call was c time
SET_ENTRY $\langle b \rangle, \langle v \rangle$	initialize the multiple entry variable b to v
SET_COND $\langle b \rangle, \langle v \rangle$	initialize the condition variable b to v
END	End of when-block

Table 5.2: Events in a when-block trace

The simulator, on the other hand, uses relative timing:

```

when-block instance A elapsed time t3-t1
send message B at t2-t1
when-block instance A ends

```

After the preprocessing, a when-block record in the trace information forms one entity. The simulator reads a when-block record at a time and processes it. The format of the entries in a when-block record is listed in Table 5.2. The events are identified by a quadruple $\langle p, b, i, r \rangle$ where p is the processor number, b is the when-block the event is happening, i is the instance, and r is the reference number.

5.5.2 Parallel machine simulator

The simulator uses an event-list based approach to simulate the machine model. An event contains event-time, event-type, and other information depending on the event type. The events are kept in a heap. There is one entry for each processor and coprocessor in the heap. Each entry contains a sorted list of events that are to happen on that processor or coprocessor. The time stamp of the heap entry is that of the earliest time event in its list. The simulator removes the next event from the heap and processes it until the heap becomes empty. The coprocessor events handle the message send and receives. They contain the necessary information about the communication including message destination, length, priority etc. The processor events are

either user events (a when-block) or system events. The system events handles the the incoming message queue in the processor. The processor continuously fetches messages from the message queue and puts into the scheduling queue. If the message queue is empty, the next message from the scheduling is fetched and processed. If that message triggers a when-block instance, then the when-block record is read from the trace file and the when-block trace is interpreted. After the when-block is finished, the priority is given to the system events to transfer messages from the message queue again.

5.5.3 Interpreting the traces

Interpreting traces requires modelling of the Dagger runtime system. The simulator has to schedule when-blocks based on the arrival order of messages in the same way Dagger does. The simulator models Dagger by maintaining three major queues: scheduling queue, wait-queue, and ready-queue. All of the incoming messages are buffered in the scheduling-queue. The management of this queue can be FIFO by default, or it may use other queuing strategies. The simulator supports LIFO, prioritized FIFO, and prioritized LIFO strategies. The prioritized message scheduling can be used in certain applications to decrease the execution time. The wait-queue is a list of when-block instances waiting for some messages or completion of other when-blocks. This queue is necessary because a when-block may depend on multiple messages or when-blocks. When dependences of a when-block instance are satisfied, it is moved from the wait-queue to the ready-queue.

When the processor fetches a message from the scheduling queue, it emulates the Dagger runtime to process the message. It checks the dependency graph to determine if any when-block instance depends on this message. If so, then it checks the wait-queue to see if the when-block instance already has been created, otherwise it creates the when-block instance and puts it into the wait-queue. If the arrival of the message causes one more when-block instances to become eligible for execution, those instances are put in the ready-queue for execution. If there are multiple when-block instances in the ready-queue, the first when-block instance from the ready-queue is executed by default. However, this queue can be managed differently along with the scheduling queue. The trace record of the executing when-block instance is fetched and the events belonging to the when-block are executed in the same sequence. Note that the trace contains events that denote sending of messages and synchronization. The local time of

the processor is incremented by the appropriate delay of each action including blockings due to network load. At the end of the when-block interpretation, dependences are inspected to detect if any other when-block instance is waiting for the completion of the completion of the current when-block.

The cost of management of the message and when-block queues are reflected in the simulation time also. The cost of receiving a message and transferring it to the scheduling queue is already represented by the σ_r in the machine model. The processing of the messages in the scheduling queue, creating a waiting when-block instance and initiation of a when-block execution, are represented by the parameters d_m , d_w , and d_s .

Chapter 6

Performance studies

In the previous chapters, the potential benefits of message-driven execution were established. Techniques to predict the performance of message-driven programs under different architectural assumptions were developed. This chapter investigates the performance benefits. In particular, the following issues are addressed here:

- When does message-driven execution improve performance?
- How much does message-driven execution help (*i.e.*, quantify the benefits of message-driven execution on a range of programs)?
- What are the factors that affect the performance of message-driven programs?

The studies in this chapter were conducted with a few synthetic benchmarks and a few real benchmarks. The SPMD and message-driven formulation of the benchmarks were compared to determine under which conditions message-driven execution is desirable.

The studies can be grouped as follows:

- real machines,
- simulation studies to determine the effect of network latency and bandwidth,
- the effect of random variations in the latency,
- the effect of a coprocessor on the performance of the selected problems.

While studying these benchmarks, it became clear that the usual criteria of minimizing the completion time and reducing the critical path that are used in algorithm design are not exactly suitable for message-driven programs. Relevant and appropriate criteria in this context are developed in the context of an example in Section 6.5.

Message-driven programs are characterized by their ability to process one of many possible messages at a given point in time. When more than one message is available at a time, which one should be chosen? This decision is embodied in the scheduling strategy. A natural scheduling strategy is the FIFO strategy. In the FIFO strategy, messages are processed in the order they arrive. However, intelligent application-dependent scheduling strategies can often improve the performance of message-driven programs further. The prioritized scheduling thus combines the benefits of adaptive scheduling (with message-driven execution) and optimal scheduling. These issues are explored further in Section 6.6.2. For certain high priority messages, interrupt-driven preemptive scheduling may be used. The potential performance impact of such scheduling is examined via simulation studies in Section 6.6.1. Conclusions derived from the performance studies are summarized in Section 6.7

In the simulation studies, the parameters that are used to describe various parts of the communication delays are as follows:

Network delay $\alpha_{net} + n\beta_{net}$

Processor overhead $\alpha_p + n\beta_p$

Coprocessor overhead $\alpha_{cp} + n\beta_{cp}$

The processor and coprocessor delays are the same for receiving and sending operations.

6.1 Description of benchmarks

This section describes the benchmark algorithms that have been selected to conduct the performance studies. The selected algorithms were implemented in message-driven and SPMD styles using the Dagger language. The traces from the machine runs of these programs were used in simulation studies. The algorithms can be grouped into four classes:

- The first group consists of synthetic benchmarks, which are prototype algorithms that contained features that create opportunities for overlap.

- The second group contains the concurrent reduction operations. Reduction operations are very common in many computations. The global communication nature of the reduction operation creates idle times in processors that cause performance degradation in parallel applications. Many algorithms with reduction operations have been modified to reduce this bottleneck. Among them, asynchronous reductions [83] and pipelined reductions [79] can be counted, as well as reducing the number of reductions by combining a few of them together [18]. Due to the widespread necessity and importance of such reduction operations, the concurrent reduction benchmark was chosen to represent this class of computations.
- The third group is the core part of a parallel implementation of a CFD algorithm - Harlow-Welch [51], in which different numerical techniques have been covered.
- Finally, another numerical algorithm, Conjugate Gradient (CG) method has been studied. The CG method also involves global, reduction-like operations. Some work has been done to decrease this bottleneck [79, 18]. We will use a model of a variant of CG method [79] (simulating its computational and communication pattern) in which the reductions are pipelined and examine what performance improvements occur if the proposed algorithm is implemented in message-driven style.

6.1.1 Synthetic benchmarks

Two benchmark programs were developed to demonstrate the advantages of message-driven execution. Although these computations are synthetic, they reflect possible practical applications (particularly large applications). The purpose of the synthetic computations is to examine and quantify the performance advantages of message-driven programs in situations where opportunities for overlap exist (in situations where dependences force a single thread throughout the computation, SPMD will clearly perform equally as message-driven execution). The synthetic computations, *Wave* and *Mlib*, are explained below.

6.1.1.1 Wave

The Wave program models a computation where every processor has some work to do by itself. From time to time, one of them needs a global operation (involving other processors and

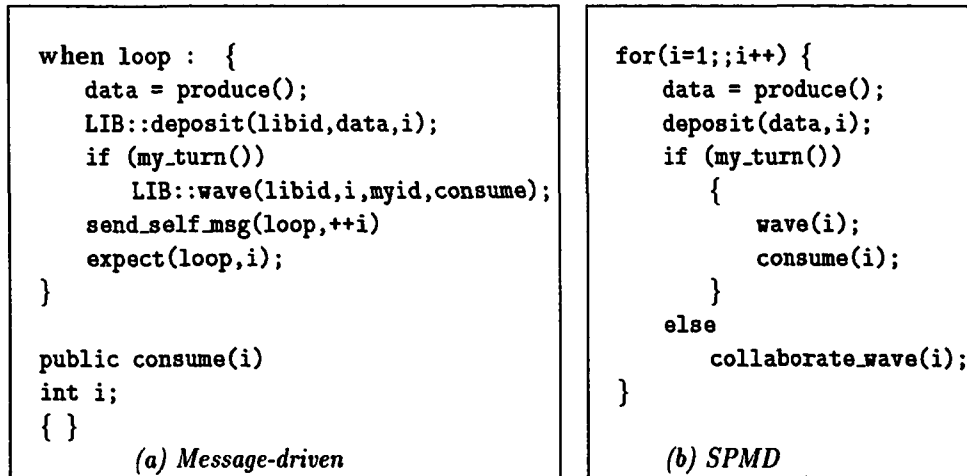


Figure 6.1: Synthetic benchmark Wave (a) message-driven (b) SPMD.

communication). In the model, each processor does some work iteratively. At each iteration, each processor produces some data. At a particular iteration, one of the processors consumes the data that can be computed by a parallel computation *M* involving all the processors in that iteration. Assuming that processors logically form a 2-D mesh and data dependences are such, computation *M* is accomplished by a diagonal wave that starts at processor 0 and sweeps all the processors. In order to propagate the wave, a processor receives two messages from it (north and west side, for instance), processes them, and sends messages out in the wave direction (south and east). This type of dependency pattern is common in many parallel computations such as triangular systems [52]. The fragments of the message-driven and SPMD code for this example are listed in Figure 6.1.

The message-driven code has a when-block that contains the code to produce and deposit data in every iteration. If it is the processor's turn to collect the data across processors, it calls a module (LIB::wave) that performs the wave operation. The call specifies necessary arguments such as module, instance, and return address as discussed in the message-driven libraries section (Section 4.3). Then the processor continues with its local work produce() for the next iteration, while the wave is going on concurrently. Note that other processors also initiate waves at different iterations. When a wave is completed, the wave module informs the initiator processor by calling the function consume(). The same computation in SPMD style

is presented in part (b). The code produces the data and deposits it. However, the interface to the wave computation is different from the message-driven case. The processor that needs the result of a wave computation initiates it and waits for its completion in order to process the result. All the other processors collaborate in the wave operation that is initiated by another processor, and then this is done by calling the function `collaborate()`. When the wave reaches the processor, the processor processes and propagates it, returns from the `collaborate` call, and only then it can continue with its own computation.

6.1.1.2 Mlib

Mlib benchmark models computations which that invokes two or more independent algorithms. Each algorithm itself has computation and communication phases. In the Mlib program, a processor performs some local computation and calls a number of library computations, each of which requires communication across processors. After all the library computations are completed, the same procedure is repeated. The library computations are independent of each other. For this benchmark, each library computation is assumed to have the same structure on each processor. On every processor, Mlib module calls the component of the library computation on the same processor. The library component (i.e., the library code that is executing on a particular processor) sends a message to another component (such that every component receives a message) and performs a local computation. It repeats the same sequence interacting with a different component and returns the result to the caller.

The partial listing of the message-driven and SPMD programs are listed in Figure 6.2 for two concurrent library computations. The message-driven program performs its local computation – `comp()` –, calls the two libraries, and waits for the result in order to continue with the next iteration. The SPMD program calls the library computations in a fixed sequence and the calls are blocking type (as discussed in Chapter 4). Notice that each library computation by itself involves no idle time except that caused by network latency (if computation phases of the libraries are uniform).

6.1.2 Concurrent reductions

Many application programs involve multiple global operations (and their associated pre- and post- computations) that are independent of each other. In traditional SPMD programs, a

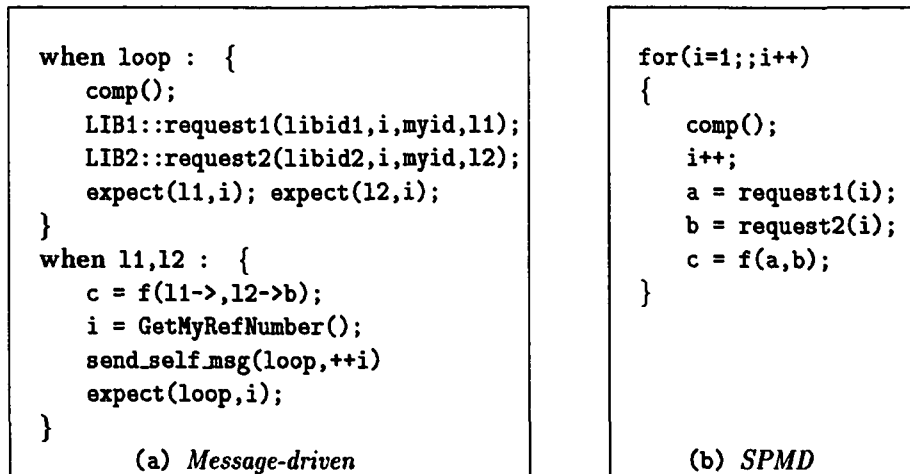


Figure 6.2: Synthetic benchmark Mlib (a) message-driven (b) SPMD.

global reduction adds a barrier: all processors must arrive at the barrier before any one is allowed to proceed beyond the barrier. Here, the barrier created by the global operations are completely artificial. It is simply an artifact of the blocking style of control transfer embedded in the underlying SPMD programming model. The next benchmark that we discuss illustrates the performance advantages of message-driven execution in this context.

This example is abstracted and modified from a real application — a core routine in parallelized version of a molecular mechanics code, CHARMM. Each processor has an array A of size n . The computation requires each processor to compute the values of the elements of the array and to compute the global sum of the array across all processors. Thus, the i^{th} element of A on every processor after the operation is the sum of the i^{th} elements computed by all the processors. In the traditional SPMD model, this computation can be expressed with a single call to the system reduction library (such as `gshigh` on Intel/Paragon) preceded by the computation of the array on every processor. Alternatively, one can divide A into k parts, and in a loop, compute each partition and call the reduction library for each segment separately.

A sample SPMD code for this computation is listed in Figure 6.3-(a). Each call to the reduction library is a blocking one, *i.e.*, the code cannot initiate the local computation belonging to the block before receiving the result of the current reduction. Therefore, in either case, each reduction call makes the processor wait until the reduction is complete.

```

        for(i=0; i<k; i++) {
            compute1(i);
            gshigh(x[i],n/k,temp);
            compute2(i);
        }

```

(a) *SPMD*

```

when next_segment : {
    compute1(i=GetMyRefNumber());
    FmaxRedn:deposit(redn_id,x,y,MyRefNumber(),reduction_done,my_boc_id);
    if (i++< k) {
        SendSelfMsg(next_segment,i);
        expect(next_segment,i);
    }
}

public reduction_done(i) int i; { compute2(i); }

```

(b) *Message-driven***Figure 6.3:** Concurrent reductions (a) SPMD (b) message-driven.

However, the computations for each partition are completely independent. In particular, the computations of the next k items (*i.e.*, the next partitions) do not depend on the result of the reduction so that they could be started even before the reduction results from the previous partitions are available. With this (message-driven) strategy, one process that has just finished computing a partition is willing either to process the result of the reduction of any previous partition or compute the next partition. Thus the wait for reduction results for a partition is effectively overlapped with the computation of other partitions. The message-driven code is listed in Figure 6.3(b). This code utilizes the concurrent reduction library. The call `deposit` initiates a reduction with the reference number i . When a reduction i is completed, the reduction module calls the `reduction_done` function which performs the post-computation. The library handles multiple reductions simultaneously with distinct reference numbers.

6.1.2.1 Estimated completion time

As the reduction involves a critical path of at least $\log p$ messages (either via a spanning tree or via a virtual hypercube-topology based algorithms), the idle time on all processors is proportional to $\log p$ and to the size of A . An approximate expression for the completion time for the SPMD model can be written as:

$$t_{\text{spmd}}^k = nt_l + (nt_r + k\alpha + n\beta) \log p$$

where t_l is the computation time per data item, t_r is the reduction computation time per data, α is the communication startup time, p is the number of processors, and β is the communication cost per data item being sent.

The time to completion for the message-driven case is given by:

$$t_{\text{md}}^k = nt_l + nt_r + kt_o + (\alpha + (\beta + t_r) \frac{n}{k}) \log p$$

where t_o is the overhead per partition in a message-driven execution model. This overhead includes context switching, scheduling, and dispatching of the messages in addition to the overhead introduced at the user level algorithm. Overlapping is reflected by the fact that the communication term is only proportional to the size of one partition ($\frac{n}{k}$) instead of the size of the whole array (n). In effect, one is required to wait only for the result of the final reduction

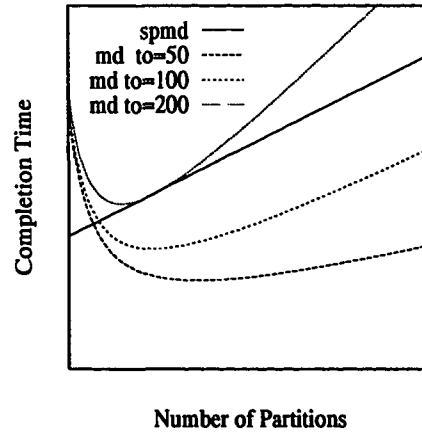


Figure 6.4: Pipelining and effects of overhead.

(for the last segment). One pays for this benefit with the overhead t_o because of the larger number of messages involved. The effect of the overhead is illustrated in the Figure 6.4. The completion time is plotted as a function of the number of segments k for various values of the overhead. The completion time for the message-driven algorithm decreases up to $k = k'$ where

$$k' = \sqrt{\frac{n(\beta + t_r) \log p}{t_o}}$$

and then starts to increase.

6.1.2.2 Actual machine results

In this section, we will present some real time performance comparison of these two schemes. First, we will show the completion time as a function of k . In Figure 6.5, the result of the reductions of size $n = 16384$ is plotted for 64 and 256 processors on ncube/2. The behavior of the algorithms closely follows the expression derived previously. For this particular case, the message-driven algorithm does better than the SPMD one-partition case (which is the best SPMD case) up to 32 partitions on 64 processors, and a little bit better for the 256 processors.

The performance benefits of the message-driven algorithm is more significant if we compare them for a fixed k . Figure 6.6 depicts the completion time versus number of processors for

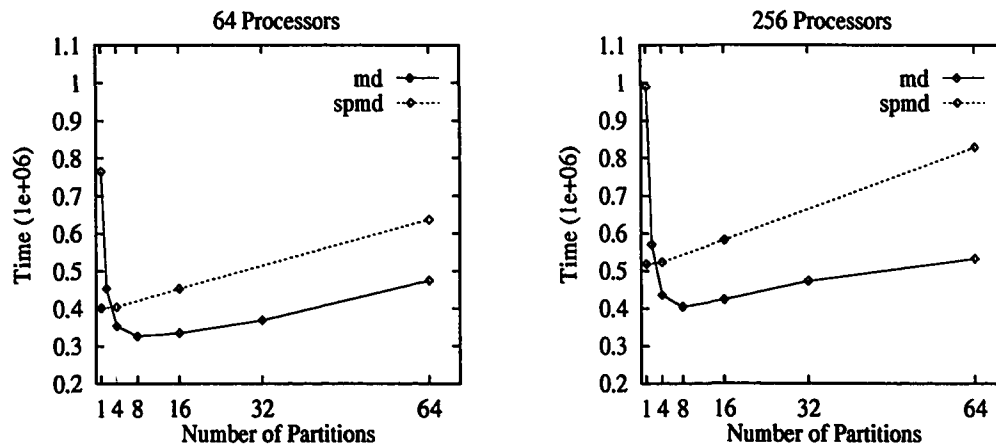


Figure 6.5: Concurrent reductions: effect of number of partitions on NCUBE/2.

$k = 8$. The problem size per processor 4096 words (n). As the number of processors increases, the reduction operations take more time, and the message-driven execution tolerates this latency effectively.

6.1.3 Harlow-Welch

Harlow-Welch is a fractional step method [64] to solve 3D unsteady incompressible Navier-Stokes equations. As a preliminary part of this research, a parallel implementation of this method was carried out, which is described in [48]. A brief description of the parallel algorithm is given here. The computational domain involves uniform grid spacing in one direction and nonuniform on the other two directions. The algorithm consists of multiple time-steps. The basic computations done at each time step are given in Figure 6.7. At each step, intermediate velocities are calculated, then FFT is applied on the uniform axis. After FFT, n independent 2D linear systems (penta-diagonal) are solved, and an inverse FFT operation is performed along the uniform axis again. Finally, velocities are corrected. The computational domain is partitioned into rectangular boxes which extend along the uniform axis as shown in Figure 6.8-(a). Due to this decomposition, FFT steps become local operations. The steps (a), (c), and (e) require communication. The communication requirements of (a) and (e) are negligible because they require nearest neighbor message exchanges only once per time-step. However, the step (c) requires much communication, because n linear systems have to be solved, which are distributed as in Figure 6.8-(b).

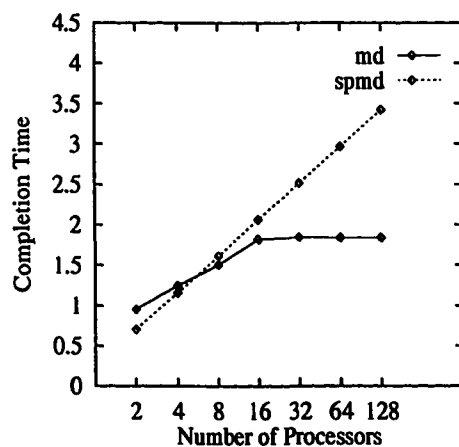


Figure 6.6: Tolerating latency: concurrent reductions on NCUBE/2.

- (a) compute intermediate velocities
- (b) perform FFT
- (c) solve n 2D systems (planes)
- (d) perform inverse FFT
- (e) correct velocities

Figure 6.7: One time step of Harlow-Welch algorithm.

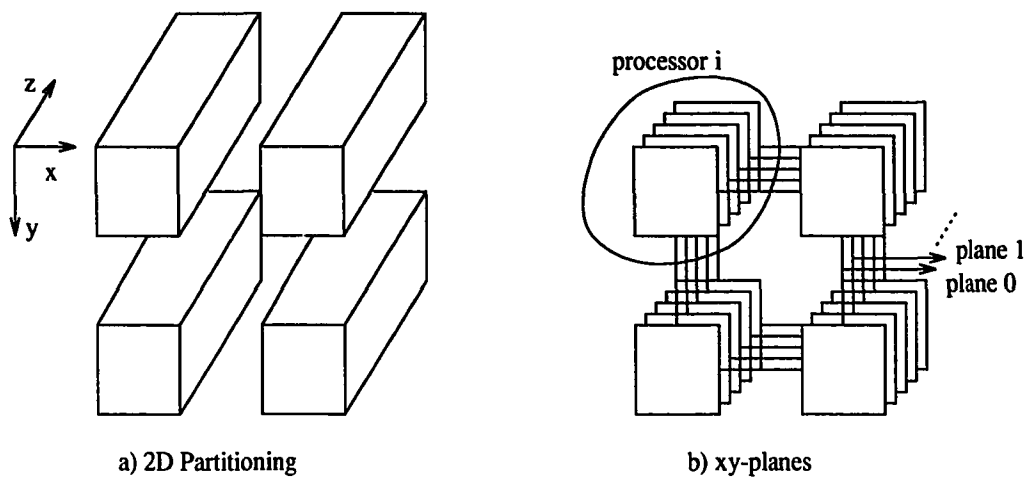


Figure 6.8: Decomposition of the computational domain.

Each of these systems are regular, sparse systems (penta-diagonal). Three methods will be considered to solve these independent planes: Jacobi, Gauss-Seidel, and Stone's method (which is a strongly implicit method). Multiple independent systems and the selected solution methods possess communication patterns common to many applications; these are nearest neighbor communication (and synchronization) in rectangular meshes, reductions, and wavelike computations (solution of triangular systems). Therefore, the result of these experiments is likely to be applicable to other computations.

6.1.3.1 Jacobi relaxation

Jacobi's method is a very common iterative scheme which is used in solving a system of linear equations. The easy and efficient parallelization of the method makes it a prototype parallel algorithm. Despite these advantages, the method's convergence rate is slow; nevertheless, its communication pattern (model) is common in many iterative solvers.

The Jacobi's method calculates the values of the unknown vector at iteration $n + 1$ by using their values from iteration n . For equations from partial differential equations discretized by the five-point stencil (the one which we have for instance), the calculation of a point u_{ij}^{n+1} involves four neighbor points. For a simple Laplace equation on the uniform unit grid for example, the value of a grid point u_{ij}^{n+1} at iteration $n + 1$ is the average of its neighbors at iteration n :

$$u_{ij}^{n+1} = (u_{i-1,j}^n + u_{i,j-1}^n + u_{i+1,j}^n + u_{i,j+1}^n)/4$$

The parallel implementation [75] is straight forward. For five-point stencil and mesh-based decomposition, each processor exchanges the boundary points with its neighbors and calculates the next iteration in parallel. This requires four neighbor messages. After every iteration a convergence is performed to determine if the solution has been reached. The convergence test requires a global operation across processors, such as `sum` or `max` type of reduction operations. The algorithm continues until the solution is reached.

In our problem, there exist multiple such systems (one for each plane). A simple implementation is to combine all the equations and apply Jacobi's method. In this implementation, every iteration basically has four phases: for all planes, a) send boundaries, b) receive boundaries, c) calculate, d) and do the convergence test. If message transmission time is longer than the


```

when boundary : {
    refnum = GetMyRefnum();
    update(Partition(refnum),boundary.messages);
    LIB:start_reduction(libid,temp,refnum,myid,recv_redn);
}

recv_redn(refnum) {
    if (not_converged()) {
        send_boundary(next_iteration(refnum));
        expect(boundary,next_iteration(refnum));
    }
}

```

Figure 6.9: Jacobi message-driven code.

time the cpu spends on sending the messages, then it must wait idly in order to continue with the calculation. Similarly, during the reduction operation, it must wait for the result of the reduction. In order to utilize these idle times, we can partition the equations and solve them in a pipelined manner. Figure 6.9 illustrates the pipelined message-driven code.

6.1.3.2 Gauss-Seidel relaxation

Gauss-Seidel method is another iterative scheme to solve systems of linear equations similar to Jacobi's method. However, the calculation of a point u_{ij}^{n+1} uses the previously computed values in the same iteration. Therefore, with the natural order of unknown points, Gauss-Seidel method is not efficiently parallelizable. For the five-point-stencil case, however, reordering of the unknowns can result in an efficient parallelization of Gauss-Seidel method. One such reordering is the red-black ordering [75].

A standard parallel implementation of the Gauss-Seidel with red-black ordering is very similar to the Jacobi's implementation. At each iteration, first red boundary points are exchanged, and each processor updates its black points. Then the black boundaries are exchanged and each processor updates its red points. At the end of the iteration, a convergence test is performed. In the Harlow-Welch context, the algorithm can be pipelined as in the Jacobi case. Figure 6.10 illustrates the pipelined message-driven code. We will call this version Red-Black1.

```

when red : {
    refnum = GetMyRefnum();
    update_black(Partition(refnum),boundary_messages);
    send_black_boundary(refnum);
    expect(black,refnum);
}
when black : {
    refnum = GetMyRefnum();
    update_red(Partition(refnum),boundary_messages);
    LIB::start_reduction(libid,temp,refnum,myid,recv_redn);
}

recv_redn(refnum) {
    if (not_converged()) {
        send_red_boundary(next_iteration(refnum));
        expect(red,next_iteration(refnum));
    }
}

```

Figure 6.10: Red-Black1 message-driven code.

Note that in order to calculate the points belonging to a particular color, not all of the four boundary messages are required. We can reorganize the grid points to introduce more concurrency (this reorganization is pictorially shown in Figure 4.8-(a), Dagger Section 4.2). Here, the grid is divided into four regions and whenever the two messages arrive that a region depends on, then that region is updated. In the black point update phase, either of quarter B11 or B22 can be calculated in any order. Then R12 or R21 can be calculated in any order (depending on message arrival times). The code for this algorithm is very similar to the previous one except that we have four when-blocks, one for each quarter. SPMD implementation of the algorithm typically specifies the execution order of the four quarters statically. A natural order would be B11, B22, R12, and R21 on every processor. However, this schedule has a problem. A particular quarter, say B11, receives boundary values of R21 from the north neighbor and boundary values of R12 from the west neighbor. Since every processor computes first R12, then R21 one after the other, B11 always waits for two-quarter update time. This can be solved with a better scheduling strategy (assuming computation and communication times are constant and

uniform). In the new scheduling technique, the order of updating the quarters is alternated for each row of processors. The even rows (the processor row number starts from zero and the topology is a 2D mesh) follow the order B22, B11, R21, and R12, and the odd rows follows the complementary order B11, B22, B12, and B21. With this schedule, the idle times created by the natural (default) order are eliminated. This version will be referred as Red-Black2 (natural order scheduling if not specified).

6.1.3.3 Stone's method

Both of the previous methods have one common characteristic; they used previously computed grid points to calculate another unknown point. In our case, this required only nearest neighbor communication. In this section, a different type of method will be applied to the equations that we want to solve. This method – Stone's method – is a strongly implicit iterative method [88]. Due to its implicit nature, the parallel implementation of Stone's method requires global communication. In order to express the computational structure of the parallel implementation of Stone's method, the sequential method itself is explained briefly below.

We want to solve a system of linear equations $Ax = b$, where A is sparse. In order to solve this system, A is factored into lower and triangular matrices L and U respectively. However, if L and U are not sparse, then solving $LUx = b$ would be computationally expensive. In Stone's method, A is replaced by $A + N$ such that L and U matrices have the same sparse structure as A (the corresponding lower or upper parts). Without entering into details of the method, the resulting iterative procedure is given as follows:

By defining

$$v^{n+1} = x^{n+1} - x^n$$

$$r^n = b - Ax^n$$

the iterative procedure can be written in the following form:

$$LUv^{n+1} = r^n$$

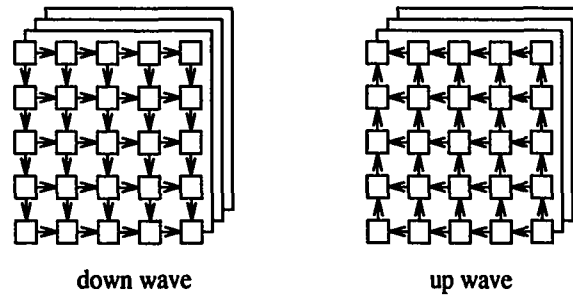


Figure 6.11: Dependences of the Stone's method.

At each iteration, we solve lower and upper triangular systems

$$Lw = r^n$$

$$Uv^{n+1} = w$$

$$x^{n+1} = x^n + v^{n+1}$$

$$r^{n+1} = b - Ax^{n+1}$$

until it converges to the solution.

Parallel implementation of Stone's method

Note that, at each iteration, a lower triangular and an upper triangular systems are solved. Given the 2D decomposition of the grid points in our problem, the dependences among the subdomains during the solution of these equations are illustrated in Figure 6.11. (The details of these equations are skipped. The important point is the computational and communication structure of the parallel method as far as understanding the simulation studies in this chapter.) The solution of the lower triangular system sweeps the processor diagonally (downward wave) as shown in Figure 6.11 part (a). The wave starts at the upper-left processor. A processor receives the boundary values from its predecessors, solves its local points, and sends its solutions to the successor processors. When the wave reaches the lower-right processor, the solution of the upper triangular system is started (up wave) as illustrated in Figure 6.11 part (b).

The utilization of processors in this algorithm is very low since only the processors on the wave are active. The existence of multiple planes creates the opportunity, on the other hand, to

pipeline the waves belonging to different planes. In the pipelined implementation, the upper-left processor solves the points belonging to the first plane and sends the boundaries immediately to its successors. Then it solves the points in the second plane while its successors work on the first plane.

The pipelined implementation can be expressed in both SPMD style and message-driven execution. Pipelining and message-driven execution are orthogonal to each other. The SPMD model can equally benefit from pipelining also. Many algorithms were improved by pipelining their computations in SPMD style [52]. However, in our case, there is one difference between these two implementations. Remember that there exist multiple (and independent) waves going backward and forward. An SPMD algorithm would typically have to specify a static order to process these waves. The natural order would be: a processor first processes the forward waves belonging to the planes $0, 1, \dots, n$ then the backward waves belonging to the planes in the same order. The message-driven style, on the other hand, adaptively chooses the waves depending on the message arrivals and message scheduling strategy. This gives a clear advantage to the message-driven execution over SPMD style for the following reason: the SPMD implementation will not propagate the back waves unless all the forward waves are processed in a particular processor. Also note that the SPMD version will not be able to exploit pipelining if the tridiagonal solver code is implemented as a library, whereas the message-driven version will not lose any performance due to the library usage. This makes the message-driven implementation even more advantageous than the SPMD one. In order to see the performance differences among these three implementations (*i.e.*, one partition, pipelined message-driven, and pipelined SPMD), expressions for their estimated completion times are derived and given below.

Let p be the number of processors, n be the problem size per processor, t_p be the processing cost per data item, t_c be the communication cost per data item, t_o be the overhead per plane, and k be the number of planes. The completion time for the one-partition case is given by

$$t_{one-partition} = (4\sqrt{p} - 2)n(t_p + t_c)$$

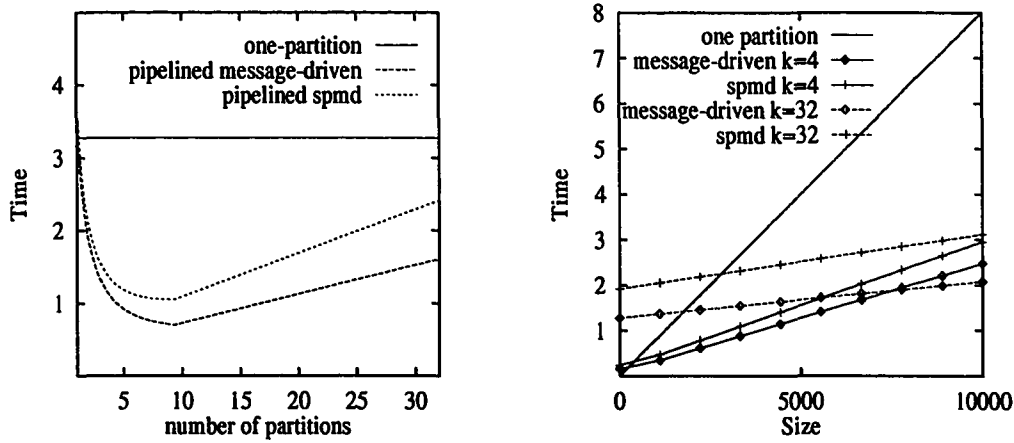


Figure 6.12: Stone's method: effect of pipelining.

the completion time for the pipelined message-driven implementation is given by

$$t_{\text{pipelined-md}} = \begin{cases} 2(nt_p + kt_o) & \text{if } (nt_p + kt_o) > \frac{n}{k}(4\sqrt{p} - 2)(t_p + t_c) \\ \frac{n}{k}(4\sqrt{p} - 2)(t_p + t_c) + (nt_p + kt_o) & \text{otherwise} \end{cases}$$

and the completion time for the pipelined SPMD implementation is given by

$$t_{\text{pipelined-spmd}} = \begin{cases} 3(nt_p + kt_o) & \text{if } (nt_p + kt_o) > \frac{n}{k}(4\sqrt{p} - 2)(t_p + t_c) \\ \frac{n}{k}(4\sqrt{p} - 2)(t_p + t_c) + 2(nt_p + kt_o) & \text{otherwise} \end{cases}$$

The estimated completion times for these three implementations are plotted in Figure 6.12. The graph (a) plots the completion time versus number of planes (partitions) for a fixed problem size, number of processors, and overhead. As it is seen in the graph, both of the pipelined versions perform much better than the one-partition implementation. In addition, the message-driven program outperforms the SPMD version. In graph (b), the completion time is plotted against the problem size (*i.e.*, the size of the domain per processor) for different values of k . The larger the k , the flatter the completion time curve. However, overhead per partition also pushes the curve up for larger values of k .

These programs were implemented and the actual performance results also measured. Figure 6.13 shows the results obtained on the ncube/2. Column (a) presents the results for the algorithms that we described above on 4, 16, and 64 processors. As the number of processors

increases, the time for a wave to complete its computation increases. Therefore the advantage of pipelined implementation becomes stronger. As the derived expressions showed also, the pipelined message-driven performs better than the corresponding SPMD program.

In the analysis and description of the computational structure of the Stone's method, we ignored the reduction operations. If we are required to perform a reduction operation for each iteration, then the message-driven program would compute a separate reduction for each plane (total k reductions). On the other hand, the SPMD program would perform one combined reduction (for the best SPMD result). If it were to compute reductions separately, it would lose the benefit of pipelining. Therefore, the overhead of k reductions increases the completion time of the message-driven program. Nevertheless, the message-driven program outperforms the corresponding SPMD in certain regions (the best performance regions at the bottom of the curves) as shown in the graphs in Figure 6.13 - (b). The partial code for the pipelined message-driven Stone's method is shown in Figure 6.14.

6.1.4 Conjugate Gradient

Conjugate Gradient (CG) method is another commonly used technique to solve systems of linear equations. Much work has been put into parallelization of CG methods on parallel machines. In a parallel implementation of a CG method, there are two types of operations that require communication: matrix-vector multiplication and inner products. Each iteration of the CG requires one matrix-vector multiplication and two inner products. In the case of sparse diagonal systems (such as the one we solve) and mesh-based data decomposition, the matrix vector products require nearest neighbor communication as in the Jacobi's method. The inner products are performed along a spanning tree similar to convergence test in the Jacobi's method.

Inner products act like a barrier between iterations and cause idle time in processors. They also cause many cache misses on the shared memory machines as they sweep the memory. Many researches have given methods to diminish the performance loss due to inner products [25]. Some of these methods attempted to reduce the number of barrier points by combining the inner products. Another way is to pipeline them as the iterations continue. Van Rosendale [79] proposed a reorganized CG method in which the inner products at iteration n are needed in the iteration $n + k$. The reorganized CG has some extra computations proportional to the

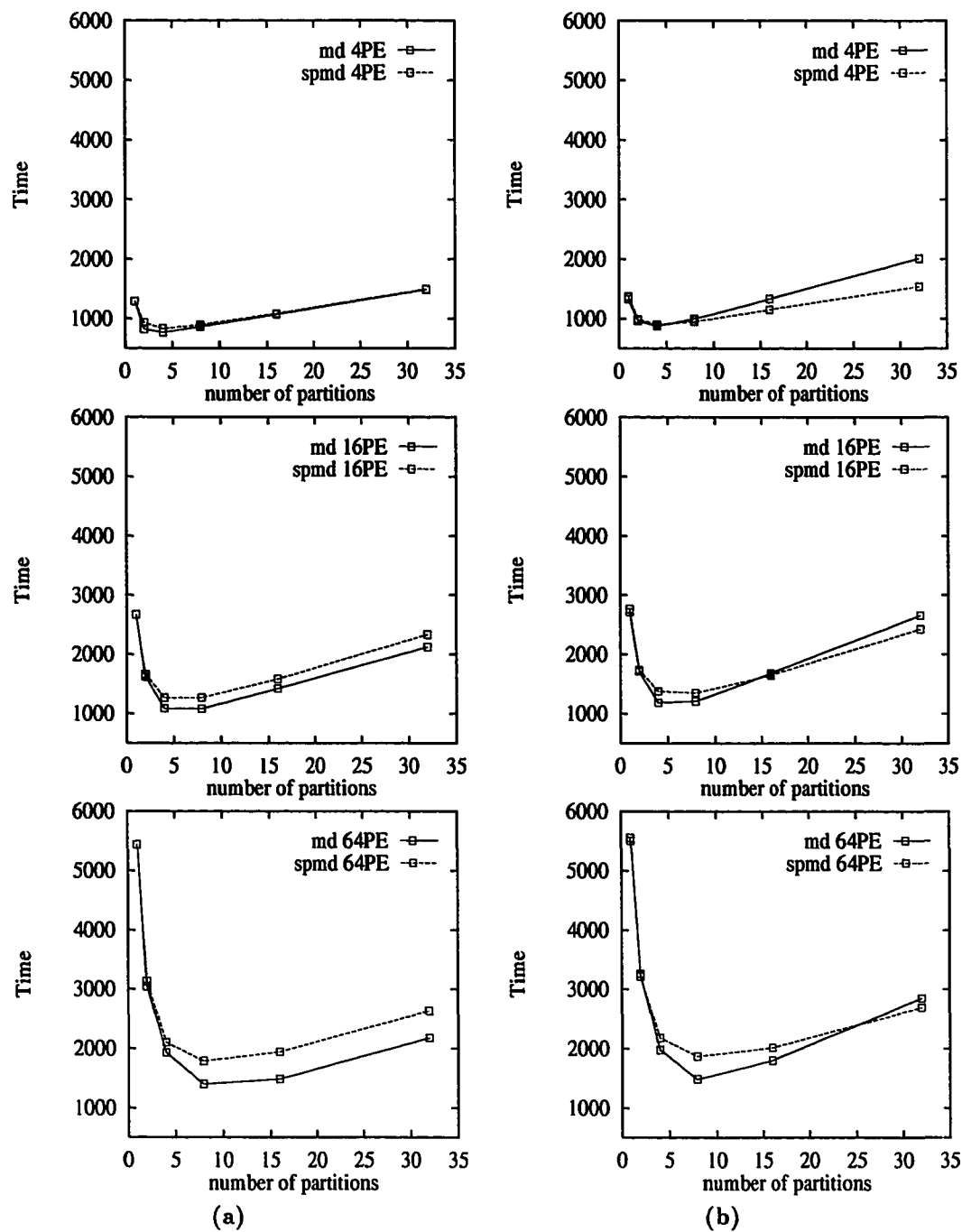


Figure 6.13: Stone's method on NCUBE/2 (times in msec) (a) without reductions (b) with reductions.


```

when down_v : {
    refnum = GetMyRefNumber();
    forward(Partition(refnum));
    expect(up_d,refnum);
    propagate_down(refnum);
}
when up_d : {
    refnum = GetMyRefNumber();
    backward(Partition(refnum));
    propagate_up(refnum);
    update(Partition(refnum));
    reflect_x(refnum);
    expect(down_x,refnum);
}
when down_x : {
    refnum = GetMyRefNumber();
    residue(Partition(refnum));
    LIB::reduction();
}
recv_reduction(refnum)
{
    if (not_converged & & first_node) {
        start_down(next_iteration(refnum));
        expect(down_v,next_iteration(refnum));
    }
}

```

Figure 6.14: Stone's method message-driven code.

Number of Processors	CG	CG-k scalar overhead			CG-k vector overhead		
		k=1	k=2	k=3	k=1	k=2	k=3
16	2219	1847	1626	1575	2178	1954	1907
64	2561	2043	1739	1652	2378	2079	1987

Table 6.1: CG results on NCUBE/2.

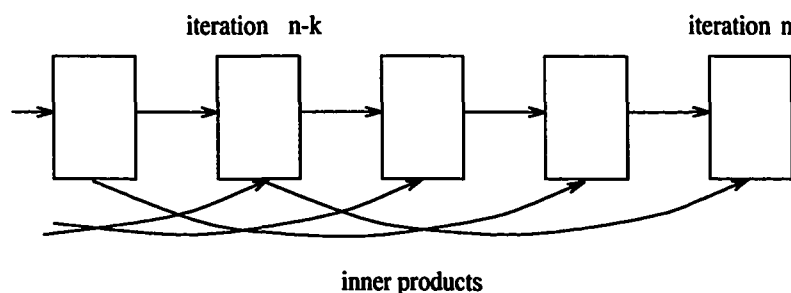


Figure 6.15: Dependences in the modified CG method.

lookahead parameter k . The author concludes that the sequential complexity of the reorganized CG method is close to the CG method (because the additional work done in the reorganized CG is scalar computations). However, it is reported that these modified CG methods may have stability or slow convergence problem [25]. Nevertheless, we will consider the k step lookahead variant of CG method, model its computational and communication structure (*i.e.*, a synthetic program which simulates a k -lookahead CG method), and investigate if such proposed algorithms benefit from message-driven execution. The dependences between inner-products and iterations are illustrated in Figure 6.15. For this purpose two programs were written¹: one performing the regular CG method and one performing the k -lookahead CG method. Table 6.1 shows the results of runs on the machine ncube/2. The problem size was 32x32 per processor and the times are for 100 iterations in mseconds. The results include the regular CG, modified CG with scalar overhead (negligible), and vector overhead (k additional inner product). As seen in the results, pipelining the inner-products decreases the completion time.

¹Note that these programs are an emulation of a CG computation

6.2 Effects of network latency

In this section, the result of simulation studies that were conducted to investigate the effect of network latency (or delay) on the performance of the selected problems will be presented. These studies were conducted on the SPMD and message-driven implementation of the problems to investigate how the two approaches handle variation in the communication latencies. The programs were run on a parallel machine and execution traces were gathered to be used in the simulation.

In order to observe the effect of communication latency, the network latency and bandwidth parameters ($\alpha_{net}, \frac{1}{\beta_{net}}$) are varied while the other parameters are kept fixed. The frozen parameters include $\alpha_p, \alpha_{cp}, \beta_p, \beta_{cp}$ and the network capacity. In the experiments presented in this section, a message of length n therefore takes $\alpha_{net} + n\beta_{net}$ time units (which can be overlapped with computation). The completion time of the message-driven and SPMD programs with k partitions (t_{md}^k and t_{spmd}^k respectively) were plotted against network latency. In all simulations, the time unit used is 100 nsec, and the simulated completion times are in 10^6 simulation units.

6.2.1 Synthetic benchmarks

6.2.1.1 Wave

Figure 6.16 plots the completion time of the Wave benchmark (t_{md} and t_{spmd}) versus network latency (α_{net}). The execution traces are from a run on 16 processors with 16 iterations. During the simulation, computations `produce()`, `consume()`, and `wave()` are assigned 10000 simulation time units. The network latency, α_{net} , is varied between 0 and 10000 units, and β_{net} is set to 1 time units.

There are two points that this graph shows. First, even in the absence of network latency ($\alpha_{net} = 0$), the message-driven Wave performs better than the SPMD Wave. This is because Wave has inherent idle times in it due to the nature of dependences and critical path (although $\beta_{net} = 1$ introduces some latency, it is negligible since the messages are very short in this benchmark). The time needed to propagate a wave one step is 10000 units (time for the `wave()` function), and 16 steps are required to complete the wave (number of diagonal blocks of 4x4 mesh). During one wave completion, only a small number of processors are busy at a time. Therefore, processors experience idle times due to the dependences in the algorithm.

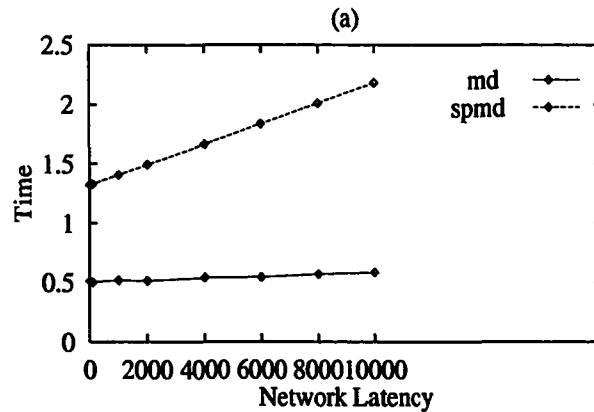


Figure 6.16: Effects of network latency: Synthetic Wave.

Note that the SPMD version has to make a blocking receive call on all processors to cooperate in the Wave computation. However, the message-driven program switches to other available computations adaptively. These include production of data for subsequent iterations and handling of other waves. In the message-driven case, there may exist multiple concurrent waves. Thus, the message-driven program interleaves the computations of multiple waves along with the computations of subsequent iterations (`produce()`).

Secondly, as the network latency increases, t_{spmd} increases at a faster rate than t_{md} because increasing the latency causes longer wave propagation. The time t_{md} also increases but as predicted in the analytical performance model in Chapter 2, its slope is less than the slope of t_{spmd} since it pipelines the waves.

6.2.1.2 Mlib

Figure 6.17 shows the completion time of the Mlib benchmark. The experiments were conducted on 16 processors with 10 iterations. The motivation behind this example is to observe the message-driven performance in the case of multiple library computations. Therefore, three sets of execution traces are gathered with one, two, and three concurrent library computations (k , number of concurrent library calls, 1, 2 and 3). Simulation experiments are performed by assigning the library computation phases 10000 time units and changing α_{net} between 0 and 10000. β_{net} is set to 1 (the messages are very short) simulation units. The effect of

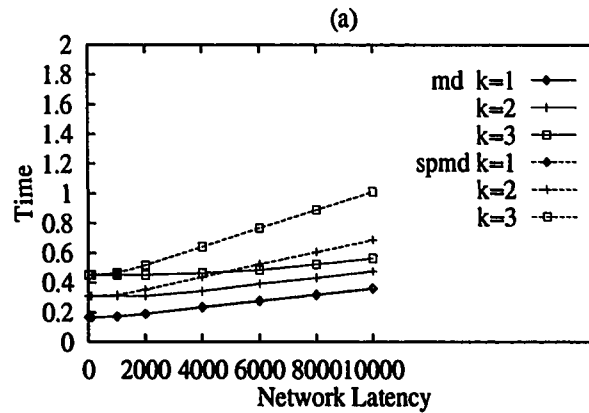


Figure 6.17: Effects of network latency: Synthetic Mlib.

the bandwidth parameter is negligible since messages are very short in this benchmark (for synchronization purposes), and therefore it is not varied.

At zero latency, the completion time of the message-driven and SPMD programs are the same for all cases, $k = 1, 2, 3$, because Processors do not experience any idle time due to communication, because the latency is zero. Also, they do not have any idle time due to processing (unlike the Wave program), because the the library computation times are the same on every processor (10000 units), and there is no load imbalance.

For $k = 1$, the completion time of message-driven and SPMD versions are equal and they increase with latency because there is no other computation that can be performed during the idle time caused by latency. For $k = 2$ and 3, the SPMD program starts performing more poorly as the latency increases as expected. The reason is that there exist 2 or 3 concurrent activities on each processor and the message-driven program overlaps the communication delays occurring in one activity by executing another available one. When $k = 3$, the difference becomes more significant for the same reason.

In the Wave problem, the SPMD program took longer than the message-driven one, even at zero latency, due to the delays created by computations on other processors. Similar delays may occur in the case of irregular or varying computation requirements. In the Mlib case, if the computation times of the libraries vary across processors randomly, then we expect the message-driven program to adapt itself to the variation and outperform the SPMD program.

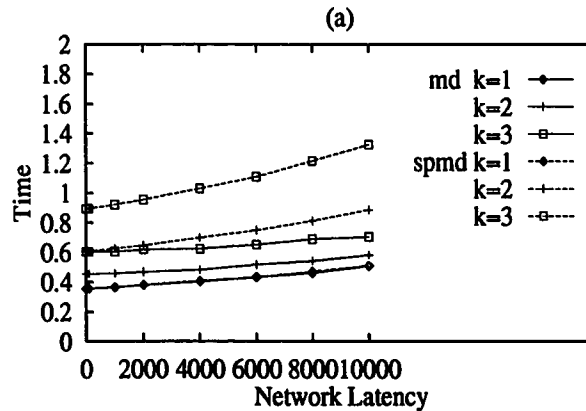


Figure 6.18: Effects of network latency: Synthetic Mlib with varying computation load.

In order to validate this observation, the same experiment is repeated by varying the times of library computations on each processor. The phases of the library computations assigned exponentially distributed times with a mean 10000. The completion time of the programs is shown in Figure 6.18. Now, at zero latency, the message-driven program outperforms that of the SPMD, because it overlaps idle times caused by delays on the other processors by scheduling the computations adaptively. Due to the variations, SPMD can not make correct scheduling decisions statically. We will show later (by examples) that further improvement in the performance of message-driven programs is possible by employing priority-based scheduling strategies.

6.2.2 Concurrent reductions

Figure 6.19 shows the simulation studies for the concurrent reductions benchmark of size $n = 2048$. The traces were gathered from runs on 16 (a) and 64 (b) processors with various values of k (number of partitions).

For different values of β_{net} , the simulated completion time versus α_{net} is plotted. Each graph depicts the performance of message-driven and SPMD programs with different numbers of partitions ($k = 1, 8$ and 64). The mean value of computation blocks (when-blocks) are 73000, 15000, and 4800 time units for $k = 1, 8$ and 64 respectively.

For $k=1$, the message-driven and SPMD programs perform equally, since there is only one reduction. For a given $k > 1$, the completion time of the message-driven program, t_{md}^k , is

always smaller than that of SPMD program, t_{spmd}^k . If we compare t_{spmd}^1 to t_{md}^k , their relative performance depends on the latency, number of processors and problem size. For this problem environment, t_{md}^8 performs better than the best SPMD program. In general, as latency increases or the number of processors increases, the message-driven program's relative performance improves (because those two factors contribute to the computation time of a reduction). For $k = 64$, however, the message creation and handling overhead exceeds the benefits of message-driven execution (except on 64 processors when $\beta_{net} = 10$).

6.2.3 Harlow-Welch

Experiments were conducted for the pressure equation part of the Harlow-Welch on 64 processors using the various iterative techniques described previously. The size of the computational grid per processor was chosen to be $8 \times 8 \times 64$ (the whole grid was $64 \times 64 \times 64$). The subdomain is further divided into k partitions (1, 2, 4, and 8). The simulated completion time of the programs, t_{md}^k and t_{spmd}^k , is plotted for various values of α_{net} and β_{net} .

6.2.3.1 Jacobi relaxation

Figure 6.20 shows the graphs of the simulated completion time against α_{net} for $\beta_{net}=0, 1$ and 10. Each graph plots the completion time t_{md}^k and t_{spmd}^k . The mean when-block time is 78000, 41000, 22000 and 12000 time units for $k = 1, 2, 4, 8$ respectively.

The t_{spmd}^1 and t_{md}^1 are the same, as expected, and they increase equally with the network latency. When we partition the domain (thus pipeline the operations), the time curve of message-driven program becomes flatter than that of the SPMD program. However, for larger values of k , the increase in the overhead due to the increase in the number of messages pushes the curve upward. For a given value of k , the performance of message-driven program is always better than the corresponding SPMD program. Also, the message-driven program does better than the best SPMD program ($k=1$) when $\beta_{net} = 10$ and $k = 2$ and 4 (or α_{net} is very large).

6.2.3.2 Red-Black1

Figure 6.21 shows the performance of the Red-Black1 implementation. The mean when-block execution is 33000, 18000, 11000 and 6600 time units for increasing $k = 1, 2, 4$ and 8. The graph plots only the message-driven completion time because the message-driven and SPMD

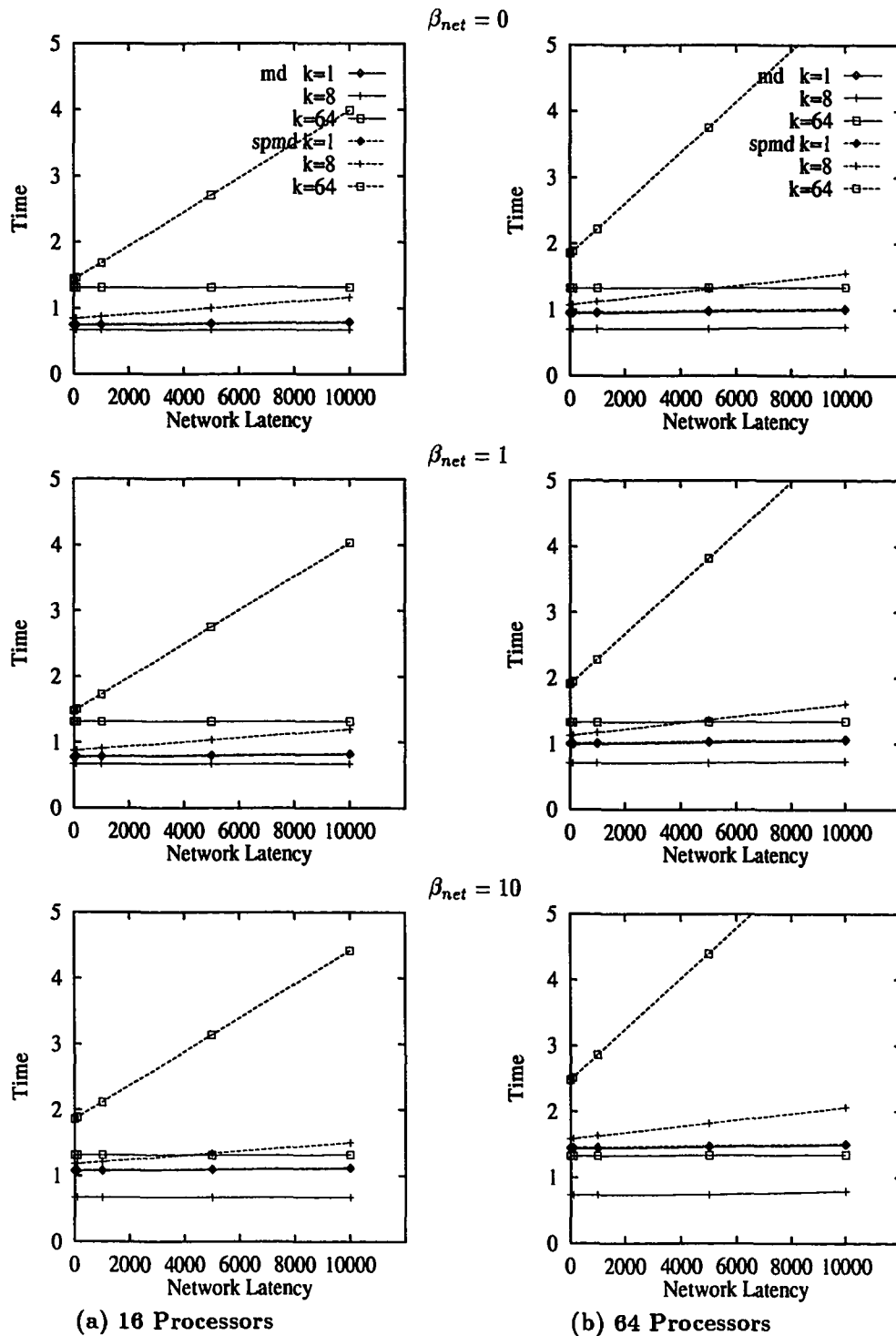


Figure 6.19: Concurrent reductions: effect of network latency α_{net} , and β_{net} .

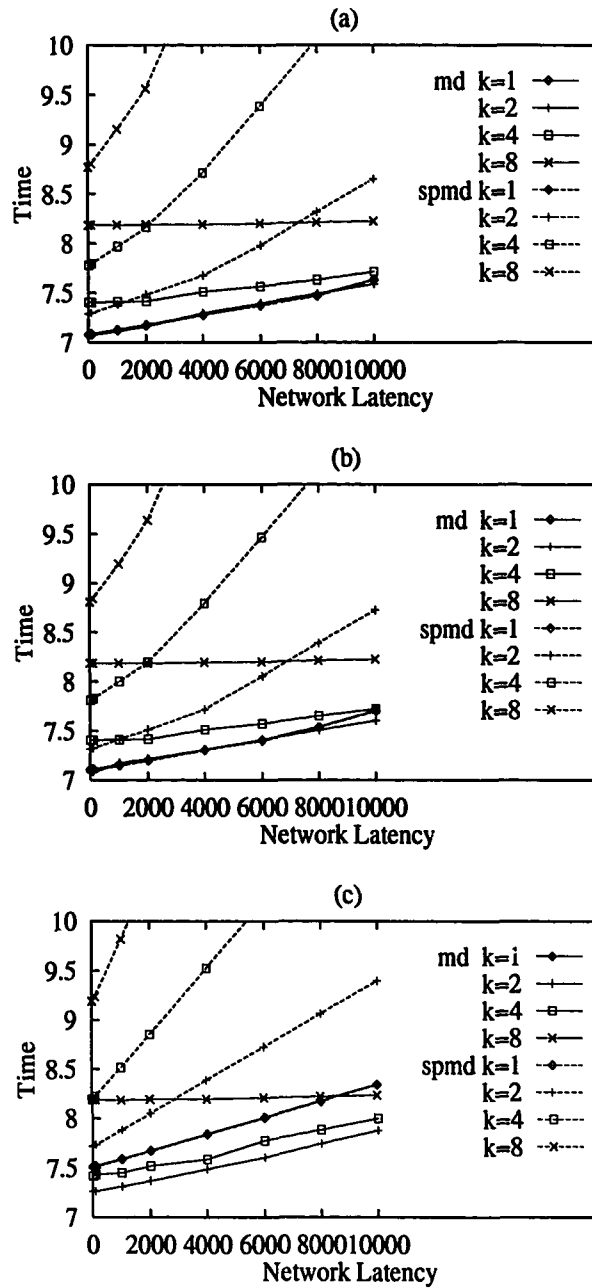


Figure 6.20: Harlow Welch with Jacobi: effect of network latency α_{net} , and β_{net} (a) $\beta_{net} = 0$ (b) $\beta_{net} = 1$ (c) $\beta_{net} = 10$.

programs perform the same for $k = 1$, and the message-driven one is always substantially better than SPMD for larger k . In these graphs, t_{md}^k will be compared to t_{md}^1 .

Under the same conditions, for $k > 1$, the curve for message-driven t_{md}^k catches up t_{md}^1 curve when $\beta_{net} = 10$ and $\alpha_{net} = 10000$. There are fewer regions where the message-driven program performs better than the best SPMD ($k=1$) program as compared to the Jacobi's implementation. The reason for this is that we are creating four more messages (at each iteration, the boundary values are exchanged in two separate messages of half size) and so the average computation time per message is decreased (when the four messages arrive, only half of the grid points are updated). Therefore, the total overhead increases (creating more messages). In addition, the ratio of overhead to useful computation increases.

6.2.3.3 Red-Black2

Figure 6.22 shows the performance of the Red-Black2 implementation for the same Harlow-Welch problem. The mean when-block time is 25000, 13000, 8200 and 5300 time units for $k = 1, 2, 4$ and 8 partitions.

Compared to the Red-Black1 implementation, the performance advantage of message-driven execution is higher in the Red-Black2 implementation. For $k=2$, the completion time of message-driven is below the curve t_{md}^1 . The reason why it is performing slightly better than the Red-Black1 is as follows: Red-Black1 updates a particular set of points, for instance red points, after receiving four of the neighbor messages. Red-Black2, however, can update half of the red points when two messages arrive, and the other half when the other two messages arrive. Therefore, Red-Black2 has better chances to utilize the idle time created by boundary exchanges.

6.2.3.4 Stone's method

Figure 6.23 shows the completion time for the Stone's method within the Harlow-Welch example. The mean when-block computation time are 63000, 33000, 18000, 10000 and 6300 for $k = 1, 2, 4, 8$ and 16 respectively.

As we discussed in the problem-description section, the advantage of the message-driven version of the pipelined Stone's method comes from the ability to handle backward and forward waves adaptively. Also, in the simulation results, the performance of the message-driven program is better than that of the SPMD program for 4 and more partitions. However, both cases

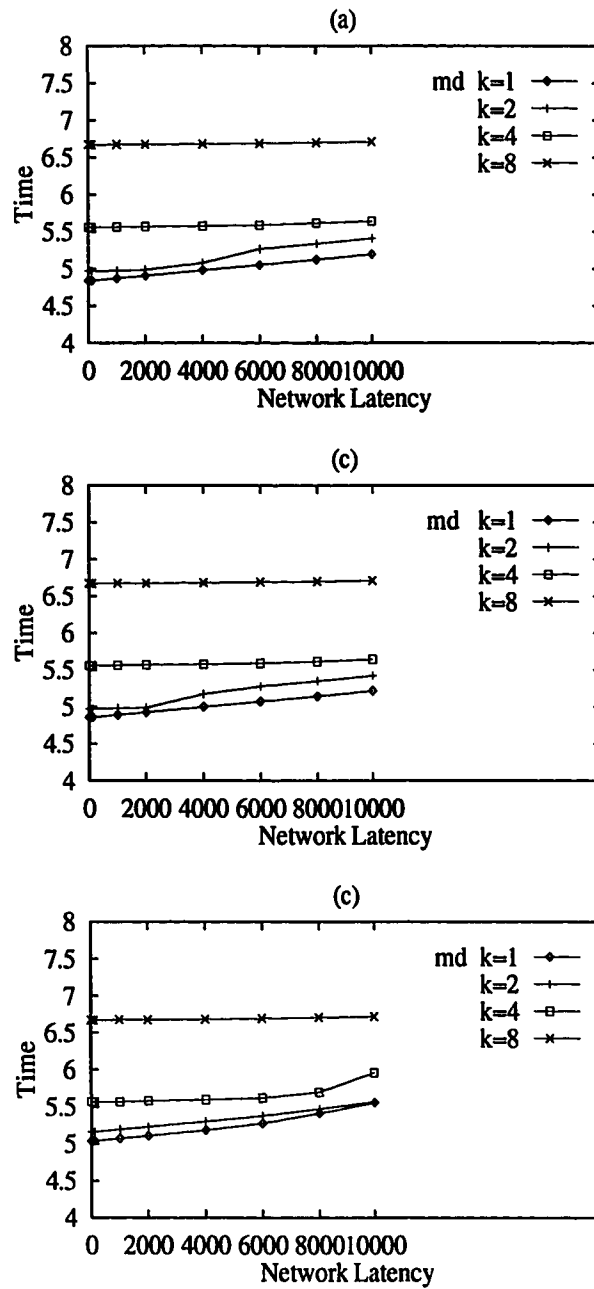


Figure 6.21: Harlow Welch with Red-Black1: effect of network latency α_{net} , and β_{net} (a) $\beta_{net} = 0$ (b) $\beta_{net} = 1$ (c) $\beta_{net} = 10$.

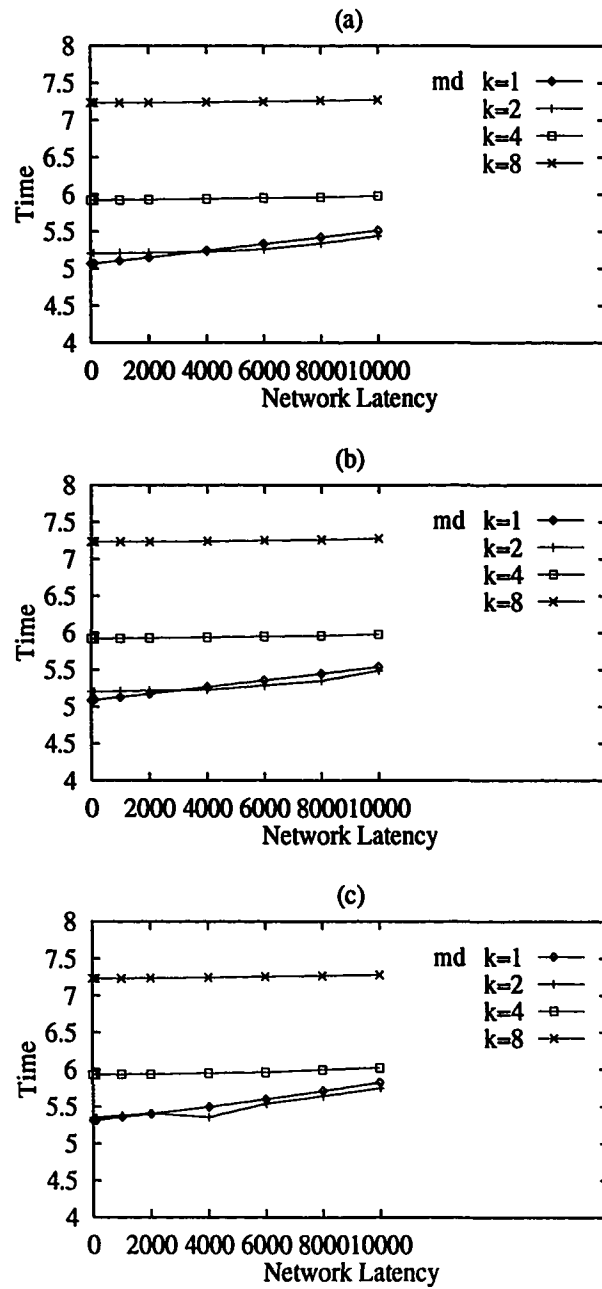


Figure 6.22: Harlow Welch with Red-Black2: effect of network latency α_{net} , and β_{net} (a) $\beta_{net} = 0$ (b) $\beta_{net} = 1$ (c) $\beta_{net} = 10$.

are affected equally by the latency increase as seen by the flatness of curves in the graphs. This is because the SPMD program also utilizes pipelining, although in a static order. (Actually, the message-driven program's curve has slightly lower slope, but compared to the whole computation time the difference is negligible. This difference will become larger for larger values of pipeline.)

Note that in all of these experiments there was a reduction for convergence checking for each plane. The message-driven algorithm does more reductions (one per partition) whereas the SPMD algorithm does one (the reductions for all planes are combined into a single array reduction operation). If the SPMD version is required to do the reduction separately, the barrier created by the reduction will destroy the ability to do pipelining and will lead to extremely bad performance. Only by assuming that the result of the reduction is not needed until the next iteration, the SPMD program can combine reductions.

Unlike the concurrent reductions (Section 6.1), the multiple reductions do not lead to performance benefits for message-driven execution because the adaptive pipelining behavior already creates adequate opportunities for overlapping computation and idle time. The k reductions simply contribute to increased overhead. The better performance of message-driven program is obtained in spite of this handicap.

6.2.4 Conjugate Gradient

Figure 6.24 shows the simulation results for the CG and modified CG method for lookahead parameter $k=1$ and 2. The grid size is 32×32 per processor. Note that this size is significantly less than the grid size in the Harlow-Welch examples. The boundary messages are very short compared these examples. As seen in Figure 6.24, the effect of latency is much more visible since the grainsize is relatively small. As expected, the pipelined version handles the delays in the reduction better than the regular version.

6.3 Effects of coprocessor

In this section, the effect of a coprocessor is studied. In many parallel architectures, a major part of the communication delay occurs in the network interfacing. The communication related processing during this phase can be performed by the processor itself. Alternatively, a

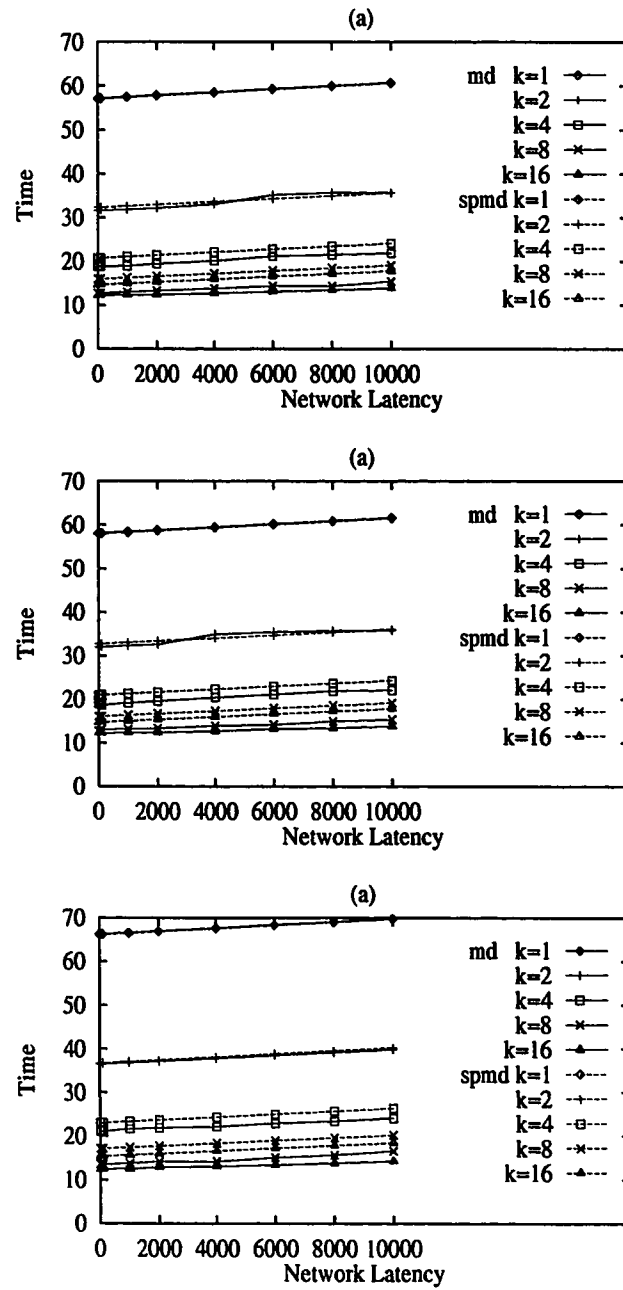


Figure 6.23: Harlow Welch Stone: effect of network latency α_{net} , and β_{net} (a) $\beta_{net} = 0$ (b) $\beta_{net} = 1$ (c) $\beta_{net} = 10$.

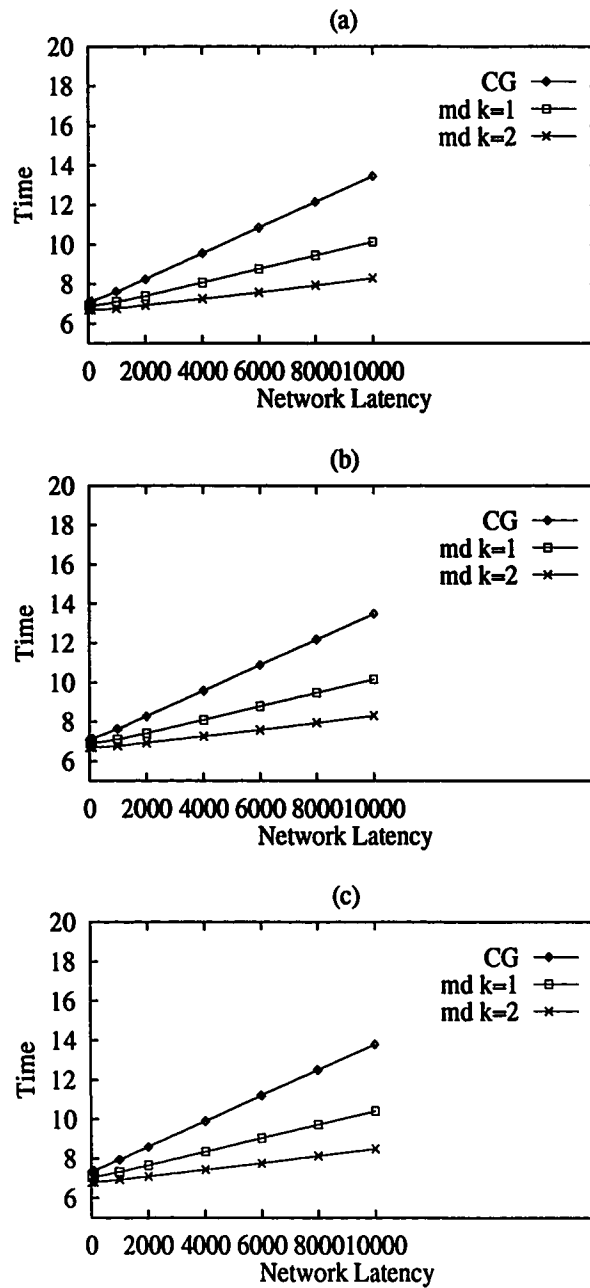


Figure 6.24: Conjugate Gradient (model):effect of network latency α_{net} , and β_{net} (a) $\beta_{net} = 0$ (b) $\beta_{net} = 1$ (c) $\beta_{net} = 10$.

Communication Parameters	Value
α_{net}	1000
β_{net}	1
$\alpha_p + \alpha_{cp}$	1000, 10000
$\beta_p + \beta_{cp}$	1

Table 6.2: Communication parameters for the coprocessor experiment.

separate coprocessor may take part of the load from the processor [57, 83, 85]. Having a separate communication processor obviously increases the availability of the processor for useful computation. Therefore, overall performance of parallel computations is expected to be better. In this section, the impact of the coprocessor on the SPMD and message-driven programs will be studied.

The experiments are conducted as follows: the network delay is kept constant. The sum of the delay which takes place in the processor and communication processor is also kept constant. However, the part of the delay that belongs to the processor and communication processor has been varied. Table 6.2 shows the values of the communication parameters used in the simulation experiments.

The simulation studies are conducted for the concurrent reductions, Jacobi's method, and Stone's methods. Figures Figure 6.25 through Figure 6.27 show the result of the simulations. The experiments are repeated for two different cases. In the first one (a), the sum of communication delays that belong to the processor and the coprocessor is set to 1000 units, and in the second one it is set to 10000 units. The completion time of the programs are plotted versus to the ratio of the delay spent in the coprocessor to the sum of processor and communication processor delay (i.e., $0 \leq \frac{\alpha_{cp} + \beta_{cp}}{\alpha_p + \beta_p + \alpha_{cp} + \beta_{cp}} \leq 1$). For total cost 1000 (a), the effect of the coprocessor is very marginal because the absolute value of the communication delay is small compared to computation time. The message-driven version benefits for $k=64$ as the coprocessor takes load of the processor.

For total delay 10000 case (b), the impact of the of the coprocessor is significant. As a larger part of the communication cost is handled by the coprocessor, the completion time decreases for both SPMD and message-driven programs. This is because the processor carries out less work. The decrease in the completion time for the SPMD case may appear counter-intuitive at first

(i.e., one may reason that message transfer time is fixed, therefore the SPMD computation will experience the delay eventually). Indeed, an SPMD program with RPC-like communications (i.e., every send is followed by an awaited response from another processor) will not benefit from the coprocessor. However, SPMD programs with consecutive sends or receives, as well as pipelined SPMD algorithms, will benefit from this because the coprocessor takes some of the processing and the processor may continue with the next computation. The message-driven programs benefit more, as shown in the results, since there exist more opportunities to utilize the time freed up by the coprocessor.

Figure 6.26 shows the result for the multiplane Jacobi example. Here, the effect of the coprocessor is more dominant (compared to the concurrent reductions) because the Jacobi program sends four more messages per partition than the concurrent reductions.

Figure 6.27 shows the results for the multiplane Stone's method. The completion time of Stone's method depends on the forward and backward waves in which the computation delays play a more important role. The effect of coprocessors is visible with 8 and 16 partitions for both message-driven and SPMD versions (slightly better for the message-driven case). Another point in the Stone's example is the $k=16$ case. With no coprocessor (i.e., the left end of the curve), the $k = 16$ case performs more poorly than the $k = 2$ and 4 cases. This is because increasing the pipeline length of a fixed computations (i.e., dividing into smaller tasks) increases the number of messages. If processor time is used to deal with these messages, the total overhead exceeds the benefit of pipelining (same as for the problem in pipelined implementation of Jacobi and other benchmarks).

For large numbers of partitions (i.e., pipelined computations), the effect of the coprocessor is significant as shown in these results. Pushing the communication cost to the coprocessor helps improve performance of both SPMD and message-driven computations. However, after some point, SPMD computations start to suffer from communication latencies, whereas those which are message-driven will not suffer as long as some other task exists.

The coprocessor may sometimes become a bottleneck (at least in transient phases of the program). The delays in the coprocessor bottleneck are analogous to remote information service delay, which message-driven execution can handle effectively.

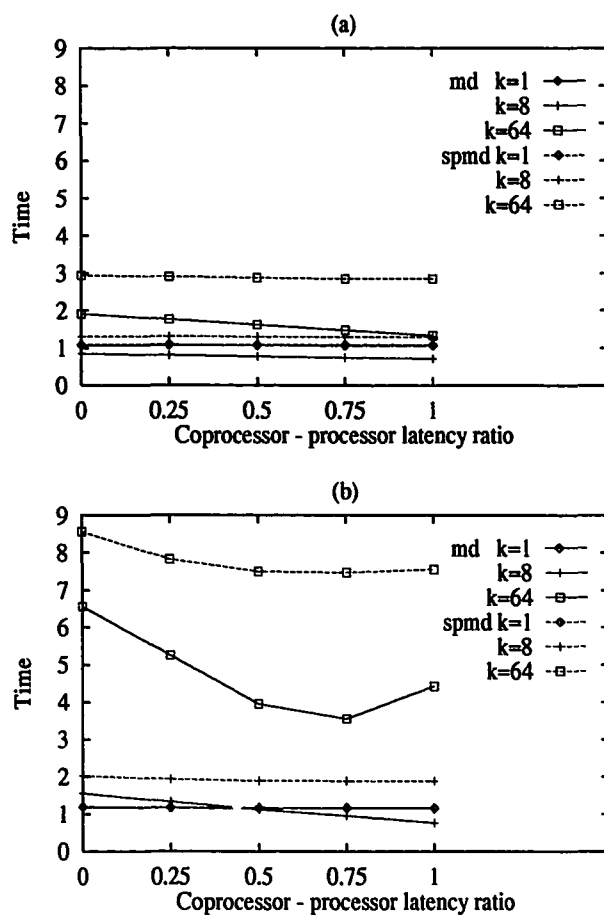


Figure 6.25: Effect of the coprocessor - concurrent reductions. Sum of processor and coprocessor delays (a) 1000 (b) 10000 units.

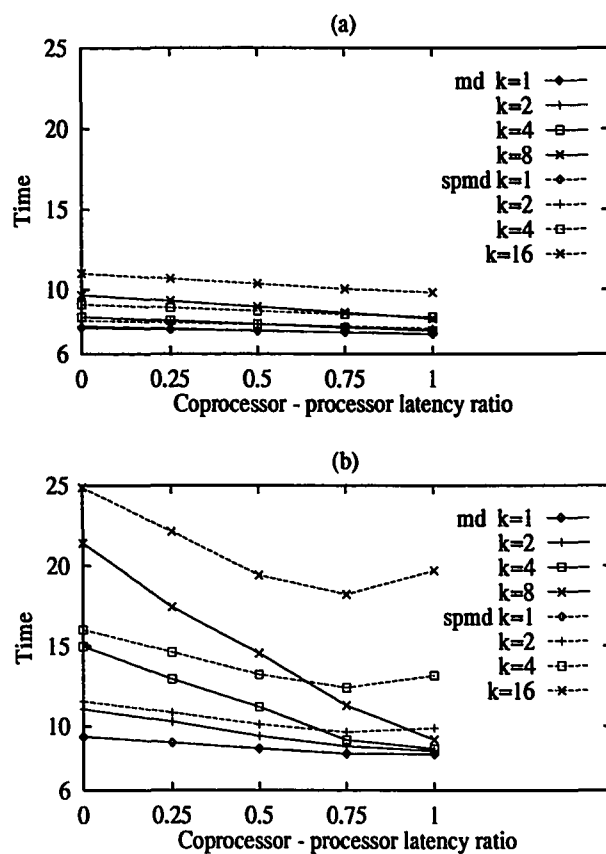


Figure 6.26: Effect of the coprocessor - multiplane Jacobi. Sum of processor and coprocessor delays (a) 1000 (b) 10000 units.

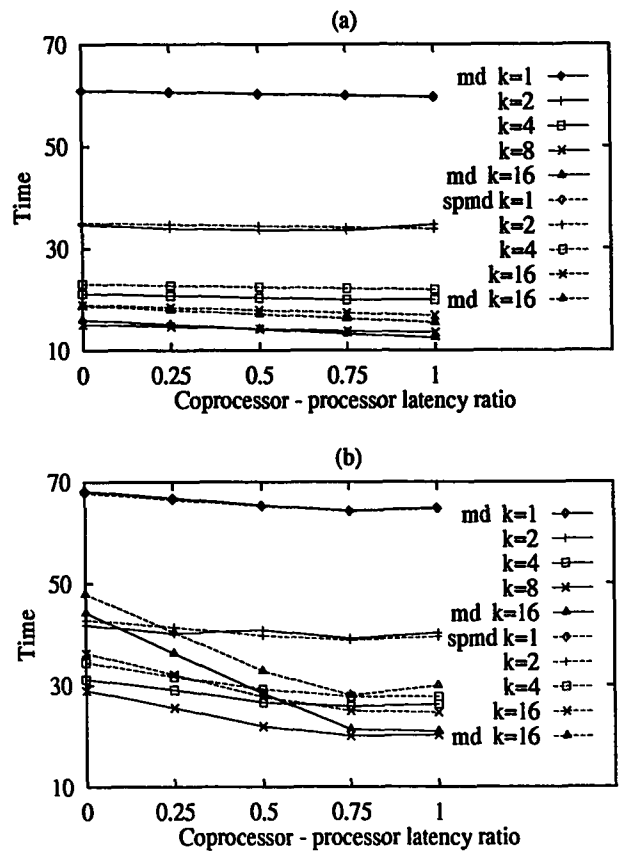


Figure 6.27: Effect of the coprocessor - multiplane Stone's method. Sum of processor and coprocessor delays (a) 1000 (b) 10000 units.

Parameter	Value
$\alpha_{net}, \beta_{net}$	1000, 10
α_p, β_p	100, 1
α_{cp}, β_{cp}	100, 0.5

Table 6.3: Communication parameter settings in the variable latency test.

6.4 Random variations in latencies

In this section, the effect of variable delays in the communication latencies will be investigated. Such delays may be caused by the load in the network; they may also be caused by other unrelated factors on machines such as networks of workstations where the network resources are being utilized by unrelated processes.

The simulation experiments are conducted as follows: the communication parameters are set to fixed values as shown in Table 6.3. During the simulation, the message transmission times are varied by adding an additional delay. The additional delay is an exponentially distributed random value. The completion time of the benchmarks are plotted as a function of average network latency (α_{net}) in Figures 6.28-b) through 6.32-b). The same experiments are repeated with constant increase (instead of random variation) for comparative purposes and the plots are shown in the same figures in part (a). Each graph plots the simulated completion time of SPMD and message-driven programs, t_{md}^k and t_{spmd}^k , for a given k . The one partition SPMD time, t_{spmd}^1 , is also plotted for comparison in applicable graphs.

Message-driven programs tolerate the variations in the network delays better. For a given k , if we compare the behavior of the message-driven and SPMD programs under constant and variable delays, we see that variation in latencies has more impact on the SPMD programs. This result is not surprising. As expected, message-driven computations utilize the additional idle times if possible.

The previous experiments were done with multiple partitions because the computations with one partition do not present any opportunities for message-driven execution. However, the message-driven Red-Black2 algorithm has concurrent computations even for one partition case (for a particular color grid points, it can execute one of the two quadrants that belong to that color). In Section 6.1.3.2, two static scheduling strategies for the Red-Black2, static

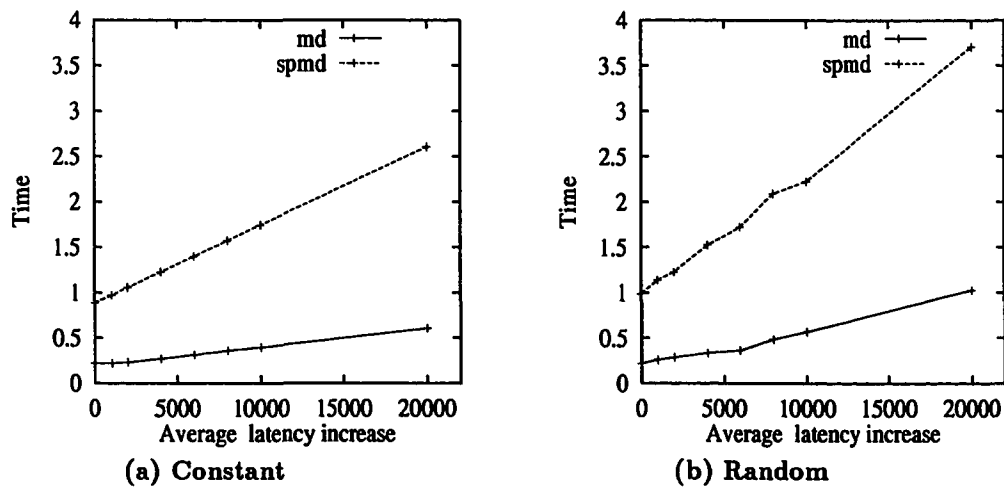


Figure 6.28: Variable network latencies – Synthetic Wave.

and optimum, have been described (under uniform latencies and computations). Figure 6.33 compares the behavior of message-driven and SPMD Red-Black2 for one partition under constant and random delays. As is shown in the graph, under constant increase in latencies, the message-driven and optimum SPMD program perform equally or better than the non-optimum SPMD program.

When variations are introduced in latencies, the message-driven program performs slightly better than the optimum SPMD program. However, the difference is small compared to the completion time. The reason might be that opportunities to be exploited by the message-driven execution are limited.

6.5 Load balance versus critical path

As has been observed so far, the message-driven execution overlaps idle times of concurrent computations successfully. This property leads to different algorithm design techniques for message-driven execution. In an SPMD computation, the idle time which a processor experiences depends heavily on the critical path of computations (which execute on some remote processor or processors). Because the SPMD model can not effectively utilize the idle time across multiple computations, the algorithms must be designed in such a way that the critical path of computations is minimized.

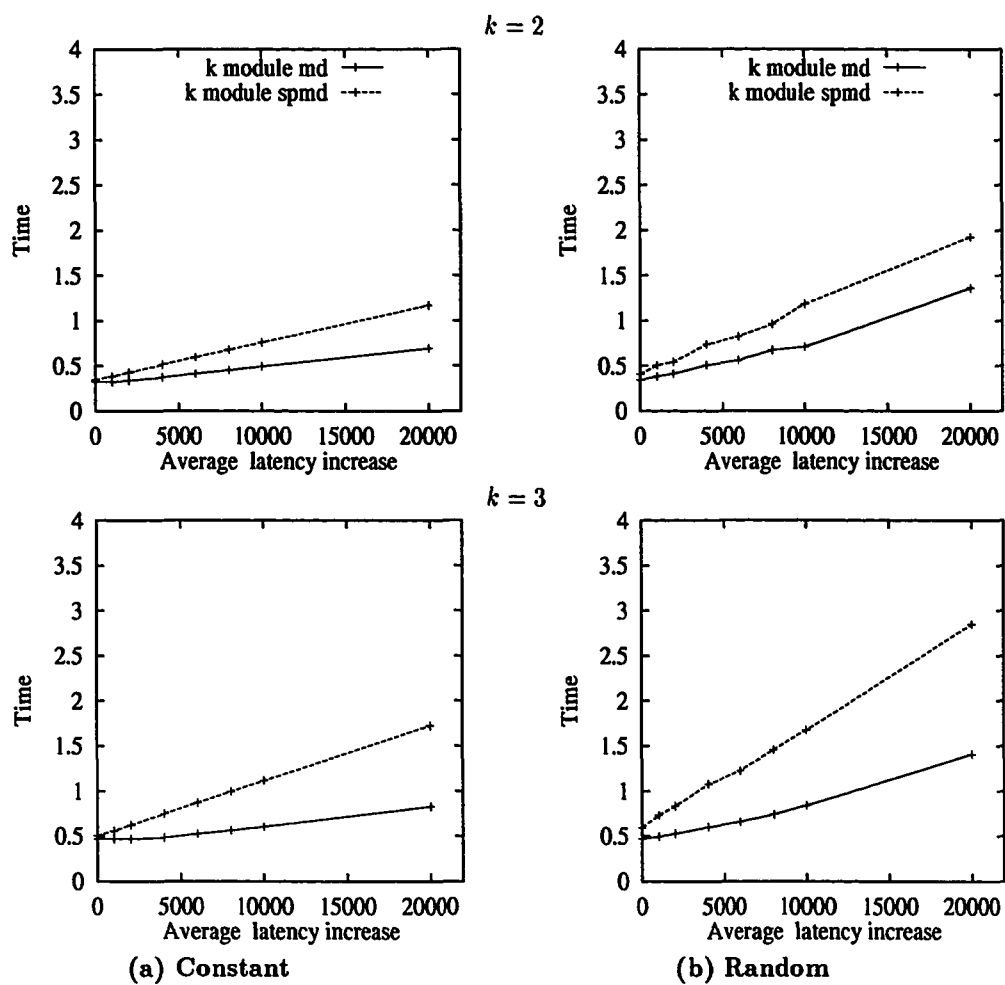


Figure 6.29: Variable network latencies - Synthetic Mlib.

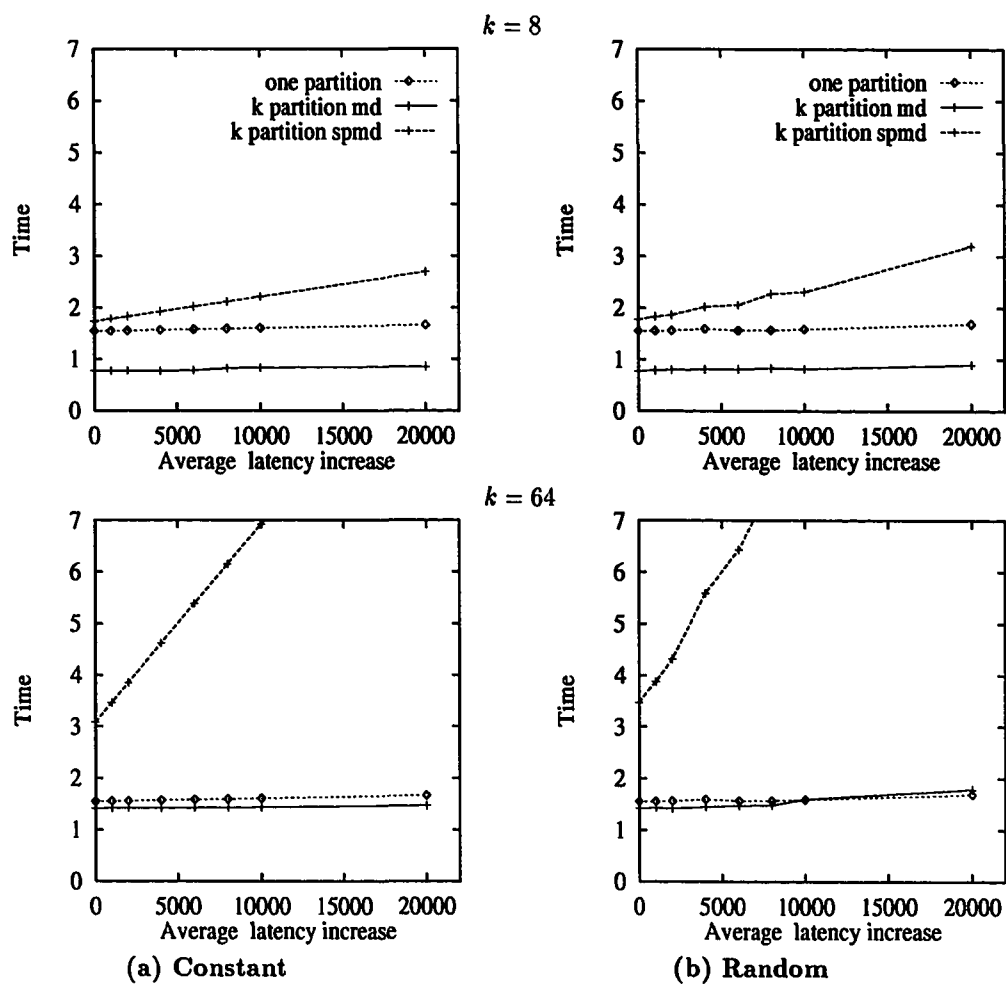


Figure 6.30: Variable network latencies – Concurrent reductions

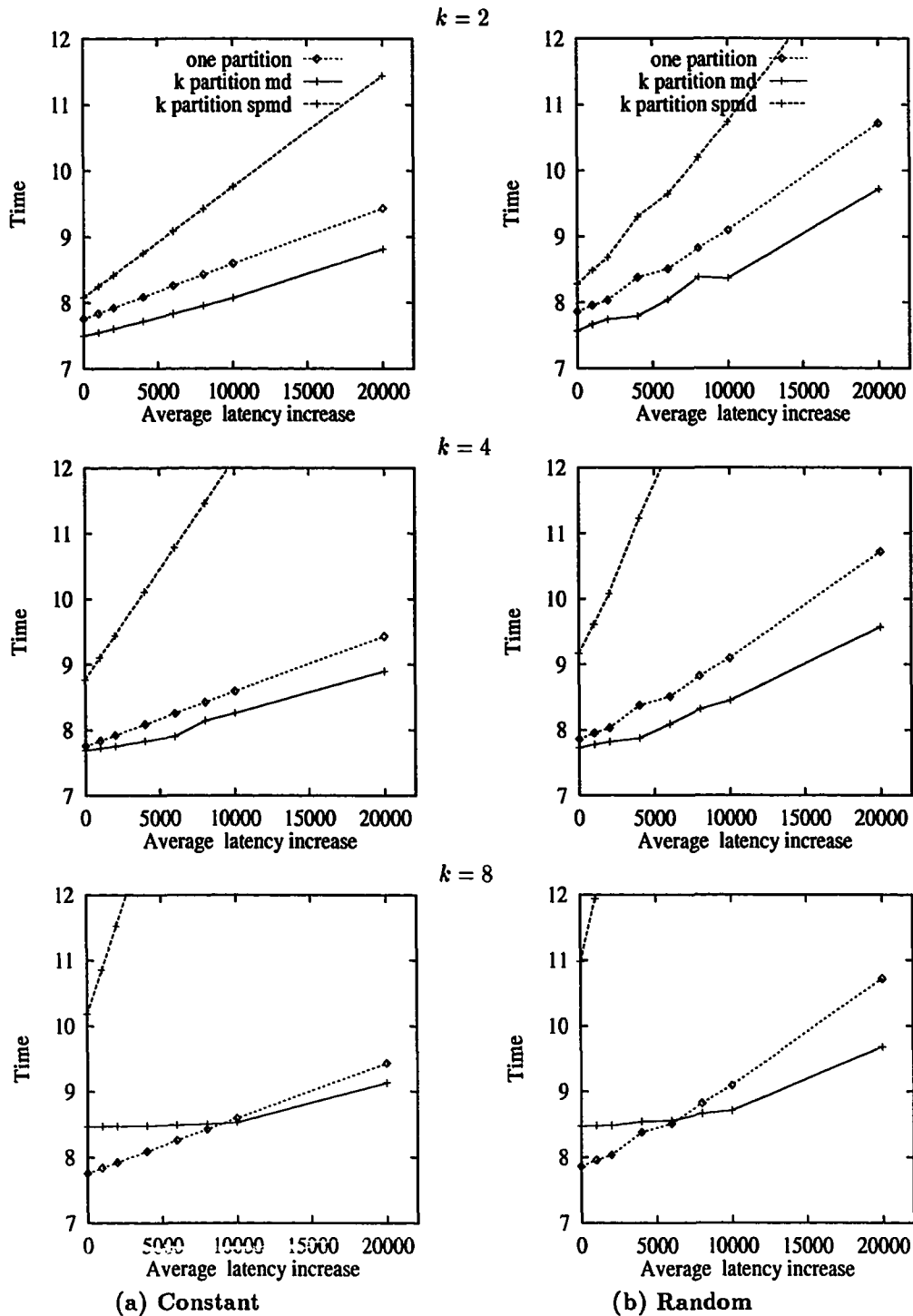


Figure 6.31: Variable network latencies – Multiplane Jacobi.

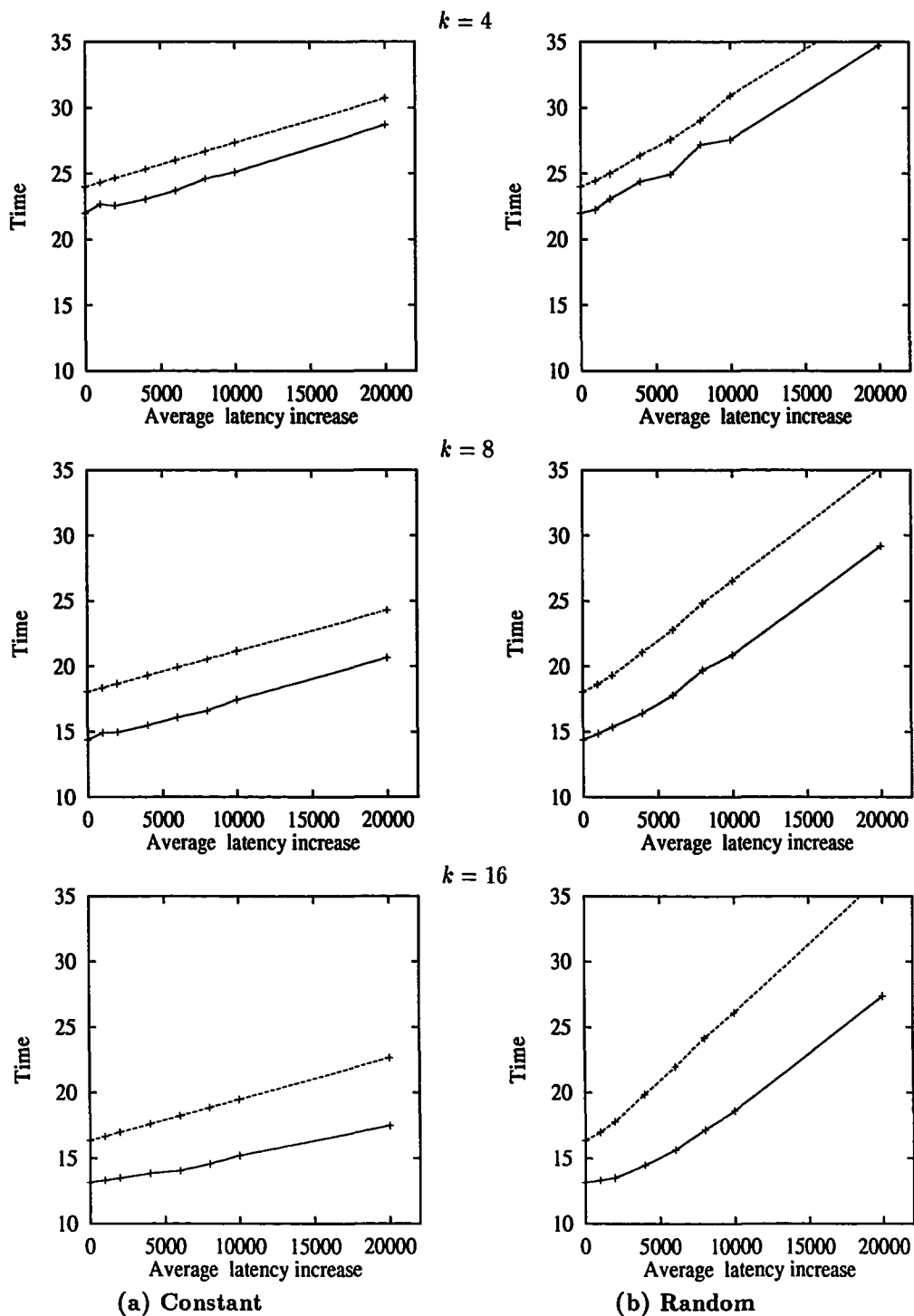


Figure 6.32: Variable network latencies - Multiplane Stone.

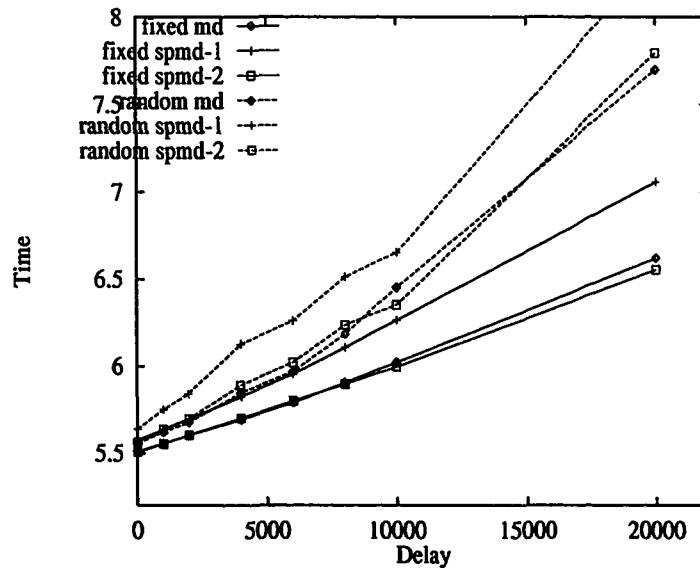


Figure 6.33: Variable network latencies – One plane Red-Black2.

However, minimizing critical paths may not be always good for message-driven programs. Consider a situation where a program can call two independent modules M1 and M2. When running in isolation, the running time for M1 is t_1 and that of M2 is t_2 . In an SPMD program, the two modules are called one after the other, and therefore the completion time for the caller is $t_1 + t_2$ (as t_1 and t_2 are critical paths in M1 and M2 respectively). It is clear that one must minimize the critical path to obtain good performance.

In a message-driven program, however, the completion time of the combined algorithm is $t_1 + t_2 - t_{overlap}$, as discussed in earlier chapters. The overlapped time depends on a variety of factors and may range from zero to $\min(t_1, t_2)$. Therefore, it is feasible to change the algorithms in M1 and M2 in such a way that the critical paths t_1 and t_2 are increased, while the opportunities for overlap are also sufficiently increased, thus increasing $t_{overlap}$ and reducing the overall completion time.

If M1 were to keep a particular processor busy for all its time t_1 and M2 also would keep the same processor busy for its whole duration, then clearly the overlapped time would be zero. Conversely, if the two modules utilize the processors in a complementary way so that the busy processors in M1 are relatively idle in M2 and vice versa, then the opportunities for overlap

are higher. Thus, load balance would seem to be another property of the algorithm that is as important as the critical path, and one can consider increasing the critical path for decrease in load imbalance.

This issue is brought to the surface by a study involving the concurrent reductions experiment discussed in previous sections. To further understand this issue, a series of experiments were conducted which are described in the next sections.

6.5.1 Load balanced spanning trees and message-driven execution

We create opportunities for a tradeoff between critical path and load balance by choosing variants of the reduction algorithms. A reduction operation is generally implemented by using a spanning tree to combine the data that reside on each processor. The spanning tree that was used in the studies was a hypercube-specific spanning tree. It guarantees that a node and its immediate descendants are its direct neighbours. The maximum branch factor on each node was 4. The work done on each processor is proportional to the number of its children. The processors are thus divided into five load groups: processors that have 0,1,...,4 children. Another possible tree is a similar tree with maximum branching factor 2. In this case, processors are grouped into three load groups. If we compare those two spanning trees, the first one has shorter critical path, but has more unbalanced load.

Experiments with these two trees were performed on $ncube/2$ on 64 processors. The concurrent reductions and multiplane Jacobi's method were executed with different branching factors. Table 6.4 shows the completion time of the concurrent reductions and the multi-plane Jacobi's program. In the concurrent reductions program, 64 concurrent reductions of 32 words in length each were performed. In the Jacobi program, the data on each processor was partitioned into 8 segments ($k=8$). As shown in the table, the completion time of the message-driven programs decreases as the branching factor decreases. On the other hand, the completion time of the SPMD programs increase as the branching factor is reduced.

Although the critical path of the spanning tree with branching factor 2 is longer, the delays on the critical path are overlapped with other computations. The more balanced load, therefore, shortened the execution time. Figure 6.34 shows the individual processor utilization (simulated) for the concurrent reduction program. The spanning tree with branching factor 2 has better load balance. The grouping of processors into classes with approximately equal utilization (as

Branching Factor	Concurrent reductions		Multi-plane Jacobi	
	message driven	spmd	message driven	spmd
4	305	566	1159	1463
3	266	591	-	-
2	230	656	1151	1576

Table 6.4: Effects of branching factor.

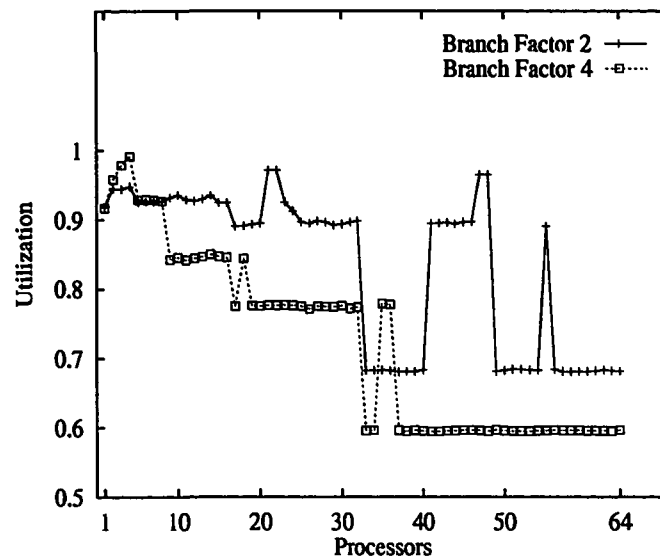


Figure 6.34: Load balance of spanning trees.

seen on the figure) corresponds to their number of children in the tree. However, there exists some load imbalance due to the leaf nodes. In the case of multiple reduction operations, this imbalance can be further improved by using complementary spanning trees.

6.5.2 Complementary spanning trees for multiple reductions

In order to further reduce the load imbalance in concurrent reductions, one can use two separate spanning trees. The leaf processors in a reduction operation can be an internal node for another reduction operation which takes place concurrently. The multiplane Jacobi program was modified to examine the effect of further load balance on the completion time. Two reduction library modules were programmed. The second one complemented the first one (*i.e.*,

Branching Factor	Hypercube ordering		Natural ordering		Complementary Trees
	message driven	spmd	message driven	spmd	message driven
4	1157	1463	1224	1512	1138
2	1151	1576	1154	1575	1097

Table 6.5: Effect of complementary spanning trees.

leaf nodes in the first one were an internal node in the second one). Table 6.5 shows the result of complementary trees for the Jacobi program. For simplicity, the complementary trees were implemented using a natural sequencing instead of hypercube specific ordering (with wormhole routing, the immediate connection is not that important in any case). The data for the hypercube ordering are duplicated from the previous table for comparison. The completion time of the multi-plane Jacobi goes further down with the complementary trees as shown in the table.

6.6 Effects of message scheduling

This section studies the effect of the message-scheduling strategies on the performance of message-driven programs for selected cases. In message-driven execution, the runtime system selects one of the messages waiting in the processor's message queues to process next. This selection, message scheduling strategy, has a direct performance effect depending on the dependences among messages and computations.

The natural (default) scheduling strategy is to select the messages in the order they arrive (*FIFO scheduling*). However, there exist some computations where the FIFO scheduling yields poor performance. Another strategy is *priority message scheduling*. In this strategy, the runtime system selects the message with the highest priority from the message queue (messages that have already arrived). If a message with a higher priority is allowed to interrupt the processor which is processing a lower priority message, then the scheduling strategy is called preemptive strategy.

6.6.1 Preemptive scheduling

Before discussing the impact of preemptive scheduling, we will present an experiment that led to need for the preemptive scheduling. The experiment uses the message-driven multi-plane Jacobi

program. This experiment compares SPMD and message-driven simulations, both based on a trace of the message-driven execution of the Jacobi program (in previous experiments, SPMD simulations were conducted on traces from SPMD executions).

The Figure 6.35 shows the result of this experiment. Simulations were conducted for various number of partitions $k = 2, 4$ and 8 , and $\beta_{net} = 0$ and 10 . The graphs plot the simulated completion time of the SPMD simulations (*fixed-order*) and message-driven simulations (*FIFO-md*) for a given k . The simulated completion time of the one-partition case, $k = 1$, is plotted for comparison reasons.

For $k = 2$, the completion time for the fixed-order and the FIFO-md are almost equal. For $k = 4$ the fixed-order simulation results in much better performance than the message-driven simulation. For $k = 8$, on the other hand, the curve for the FIFO-md is lower and flatter than the curve of the fixed-order. The results for $k = 2$, and $k = 8$ are as expected. However, for $k = 4$, it seems that the scheduling decisions made during the message-driven simulation are not good.

A reasonable explanation for this behavior is that the reduction messages are delayed by the computations (computations performed after receiving the nearest neighbor messages to update the local points). In the case of $k = 8$, the performance of the FIFO-md is better because the grainsize of these computations becomes small enough so that the reduction messages are processed properly. In order to test this explanation, the simulator was modified to allow certain messages to interrupt the processor. When a message interrupts the processor, the processor suspends the current computation and processes the interrupting message. After the message is processed, it resumes the suspended computation.

The same experiment is repeated with the reduction messages having the interrupt capability. The simulation results are shown in Figure 6.36. For $k = 2$ and 4 , the completion time of the *with-interrupts* is much better than that of both FIFO-md and fixed-order simulations. In addition, it tolerates the idle time better (the curve is flatter). For $k = 8$, with-interrupts and FIFO-md perform equally (because there are no large computations to hold reduction messages).

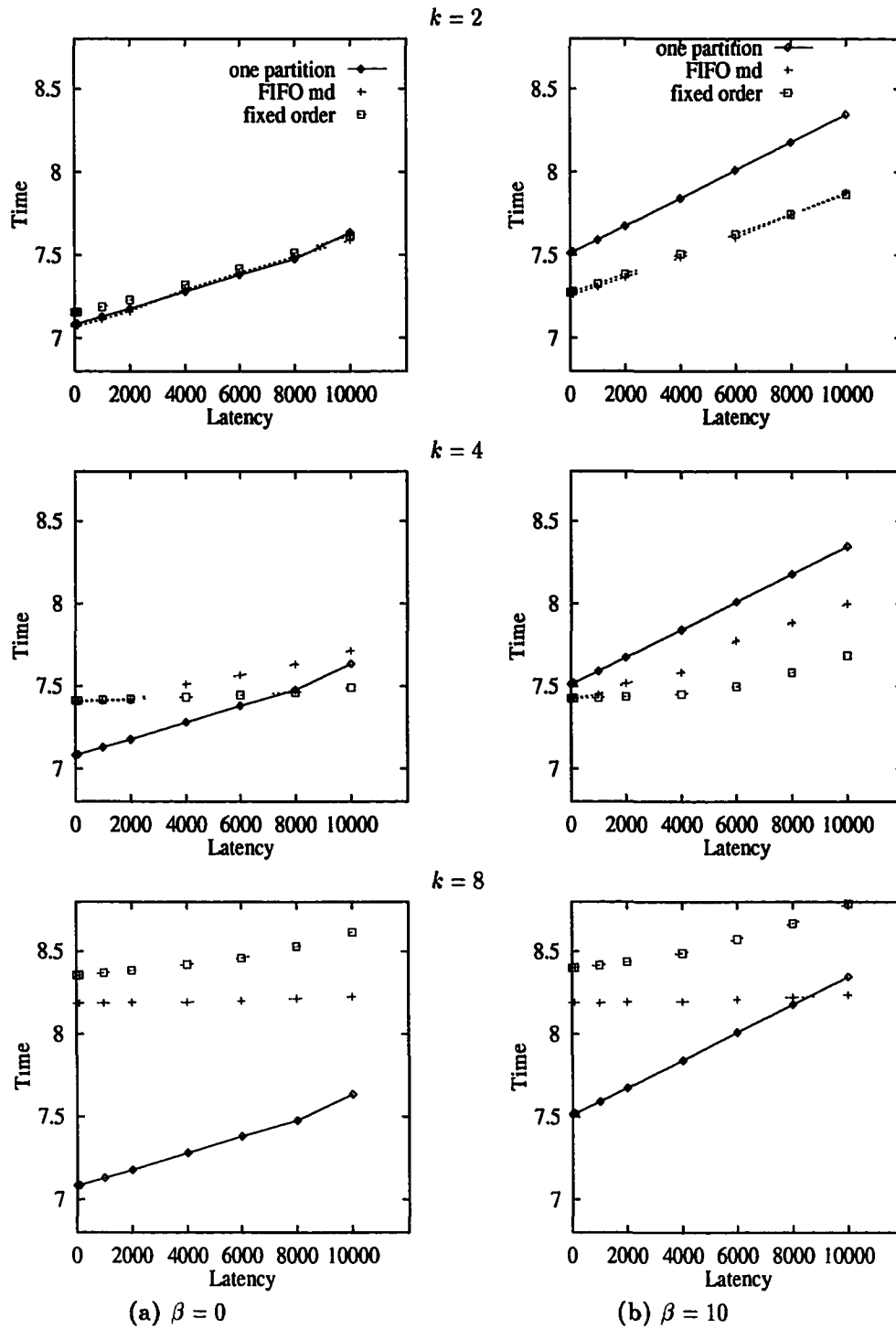


Figure 6.35: A scheduling problem

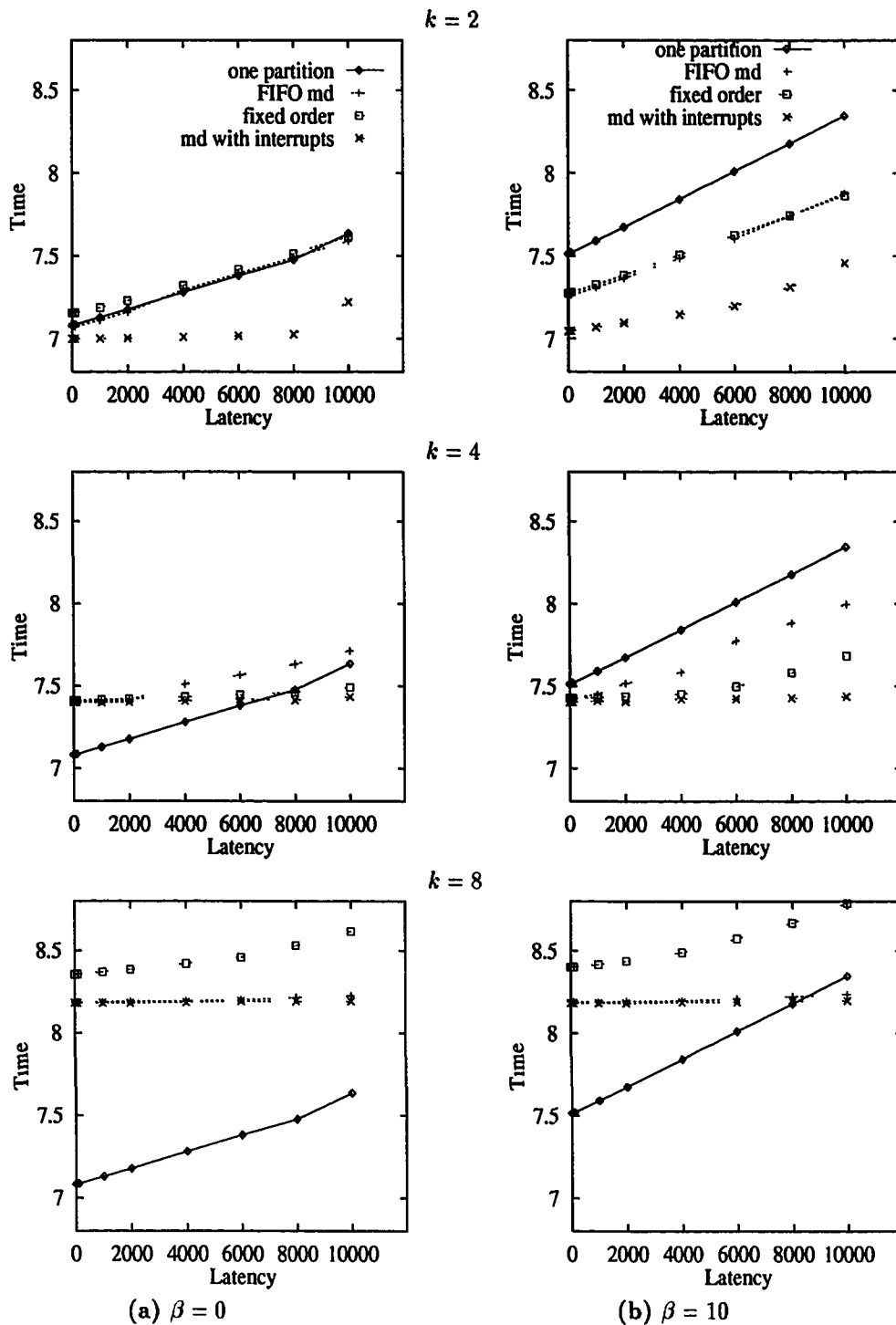


Figure 6.36. Reductions with interrupts

6.6.2 Priority-based scheduling

In this section, priority-based message scheduling will be examined. First, the previous example (interrupts) will be reexamined, and the same problem will be treated with the priority-based scheduling. Interrupts, in general, make programming more difficult (necessity to use locks) and introduce more runtime overhead (context switching). In this respect, priority-based scheduling is more preferable. Secondly, the priority scheduling will be studied in another example. This example involves the Harlow-Welch program. In the previous Harlow-Welch simulation studies, the number of iterations across planes was assumed to be the same. However, the number of iterations are often nonuniform across planes as reported in [48]. In this case, the performance of the message-driven execution (with FIFO scheduling) can be improved using prioritized messages.

6.6.2.1 Interrupts example revisited

The simulation studies of Section 6.6.1 are repeated by using priority-based scheduling. The reduction messages are given higher priorities than the boundary messages (nearest neighbor boundary exchanges). Now the reduction messages will be processed first in the case that both reduction and boundary messages are in the processor message queue, but they will be delayed if the processor is executing some user computation (even if that computation belongs to a lower priority message).

The results are shown Figure 6.37. The simulated completion time of the FIFO-md, priority message scheduling *priority-md*, and with-interrupts execution of the traces are plotted. For $k = 2$, the priority-md performs equal to the with-interrupts until the communication latency reaches a certain value. After that point, it deteriorates and performs similar to the FIFO-md. For $k = 4$ and 8, priority-md and FIFO-md perform equally. These results show that priority scheduling can be used to approximate interrupts for most cases.

6.6.2.2 Variable number of iterations and prioritized scheduling

Before discussing the application of priority message scheduling on the Harlow-Welch program, the problem with nonuniform iterations across the multiple-planes will be demonstrated with a simple example below.

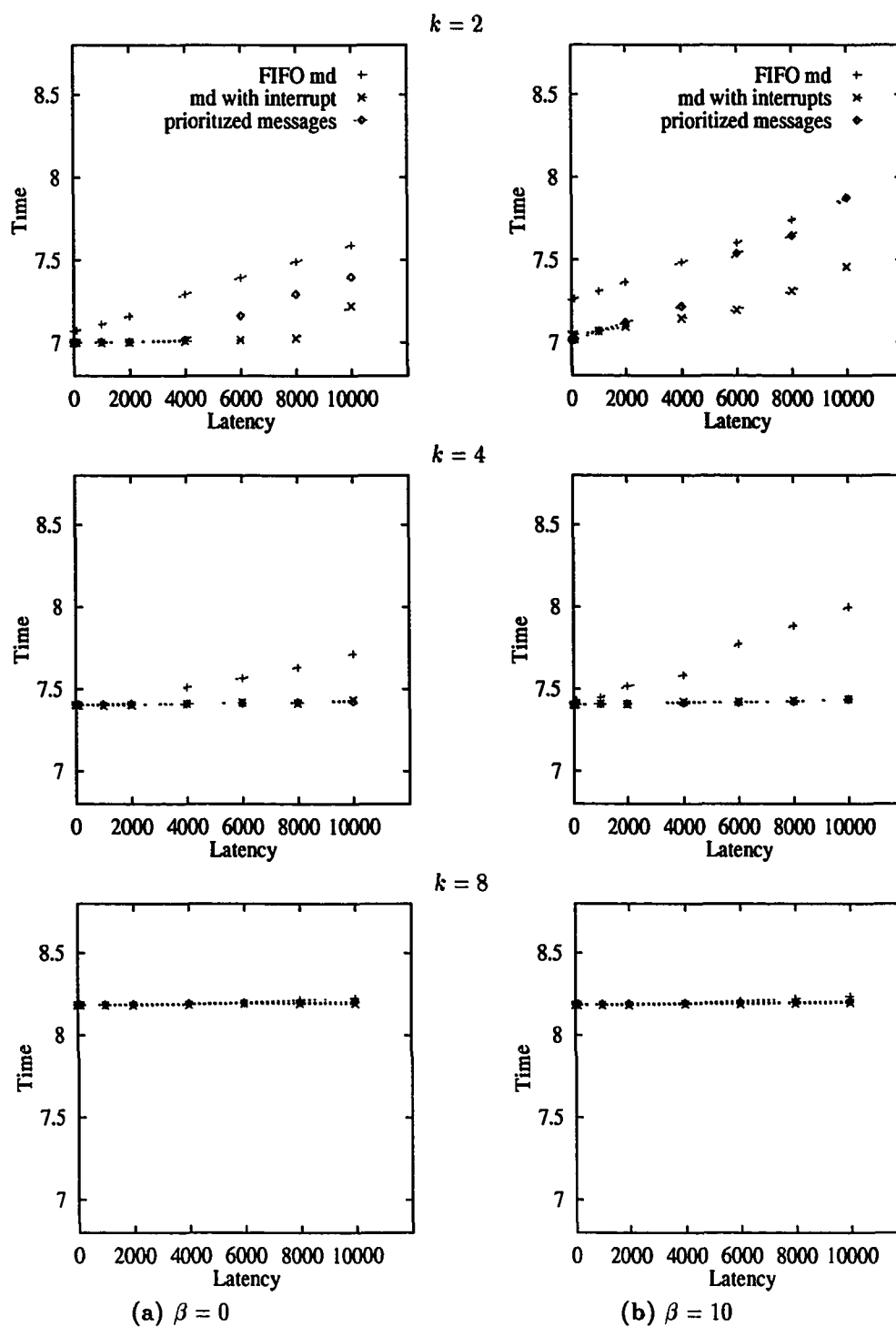


Figure 6.37: Reductions with higher priority

Assume that there are 4 planes. The number of iterations required in each plane is 4, 1, 1 and 1 (for the planes 1, 2, 3 and 4 respectively). Further assume that the computation per iteration is t time and it is uniform. Finally, let t is long enough for messages belonging to the next iteration to arrive.

Figure 6.38-(a) shows the FIFO scheduling for this example (assuming that all the planes are ready for the first iteration). The chart in the figure depicts the order of planes processed on a processor. In FIFO scheduling, the processor processes first iterations of planes 1, 2, 3 and 4. After that, only the first plane remains with 3 more iterations. The processor continues to execute iterations of plane 1. However, after every iteration, it has to wait idly for messages belonging to the next iteration. It cannot utilize this idle time, because there is no other computation that can be done during the idle time.

Figure 6.38-(b) shows another schedule for the same problem. This time, the messages for the first plane are given higher priority than the other messages. The processor processes the first iteration of plane 1 and plane 2. During the computation of plane 2, the messages arrive for (plane 1, iteration 2). After the processor has completed plane 2, it selects the first plane instead of the third plane and executes the second iteration of plane 1. As shown in the time chart, this schedule has no idle time, because there is always another computation to utilize the idle time.

The same situation exists in the Harlow-Welch example (the number of iterations are nonuniform because they converge at different rates). A series of experiments was done on the multi-plane Jacobi and multi-plane Stone's method with nonuniform iteration loads. This load information is from a real implementation [48] for the Jacobi case, and is expected to apply to the Stone's case. Experiments were made with both FIFO scheduling and priority message scheduling. In the priority scheduling implementation, the priorities of the messages are calculated as follows: the messages of a plane i gets higher priority than the messages of plane j , if i has more remaining iterations than j , based on predicted number of total iterations for each plane.

Figure 6.39-(a) shows the number of iterations per plane for the multi-plane (8 planes) Jacobi example. Figure 6.39-(b) shows the number of active planes versus time for both FIFO and priority-md case. The message-driven program was run on ncube/2 on 64 processors. As seen in Figure 6.39-(b), the concurrency in the FIFO-md scheduling starts to decrease before that of priority-md. The completion time of the FIFO-md is larger (although only slightly)

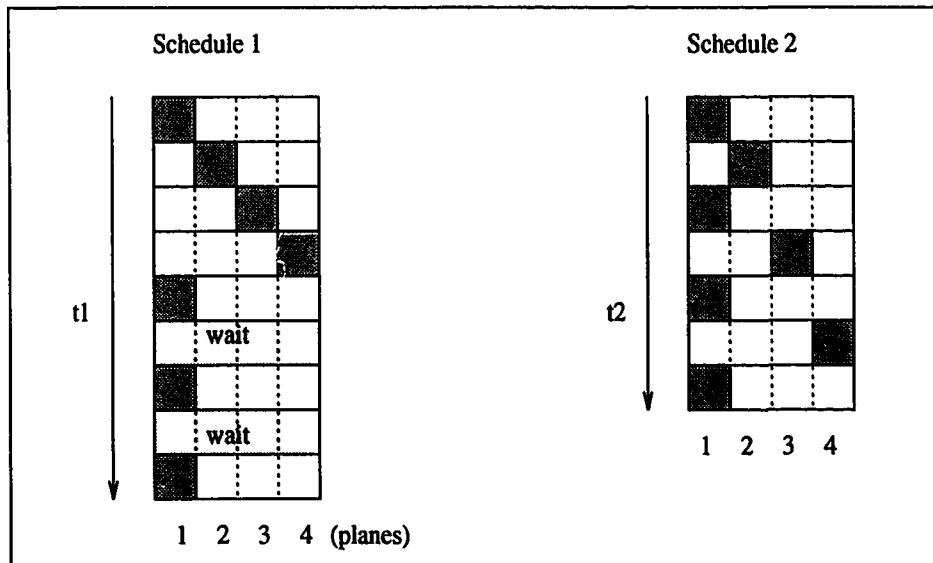


Figure 6.38: Two different schedules.

than the priority-md due to the lack of concurrency towards the end. The traces from these runs are also simulated. Figure 6.40 shows the effect of latency on both scheduling techniques. As shown in the graphs, the priority-based execution performs better. As the network latency increases, the priority-based version utilizes better the idle time caused by the latencies (the slope of the curve is lower).

The same experiment was repeated for the multi-plane Stone's method. Figure 6.41 shows the number of iterations per plane (total 16 planes). Figure 6.41-(b) shows the number of active planes versus time for both FIFO and priority scheduling case. Again, the concurrency in the priority-based message scheduling is higher and the completion time is shorter. The traces from these runs were also simulated. Figure 6.42 shows the effect of latency on both scheduling techniques. As shown in the graphs, again the priority-based execution performs better. As the network latency increases, the priority-based version utilizes the idle time caused by the latencies better (the slope of the curve is lower).

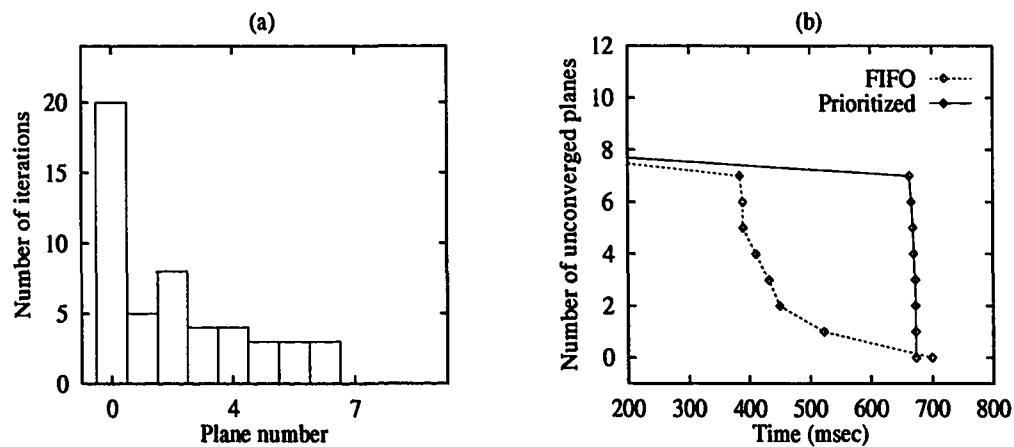


Figure 6.39: Multi-plane Jacobi (a) load index (b) concurrency index.

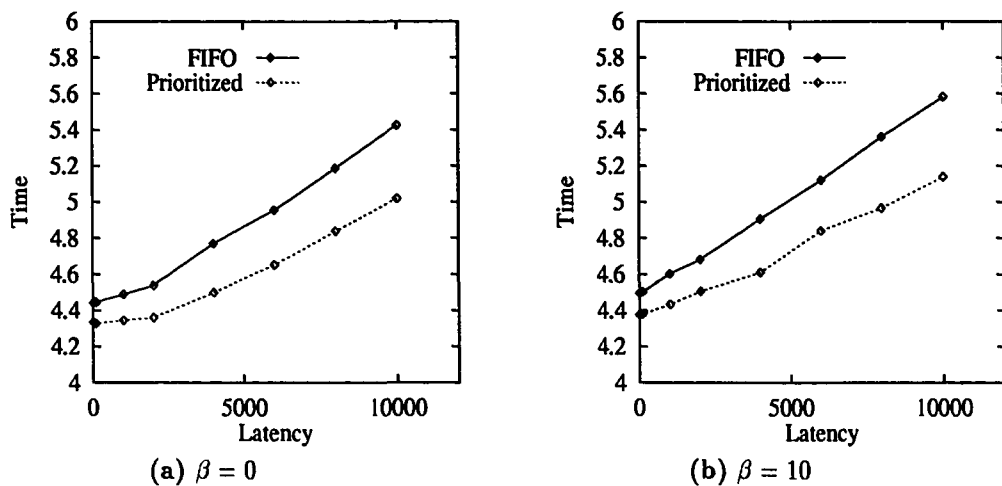


Figure 6.40: Multi-plane Jacobi : FIFO versus priority.

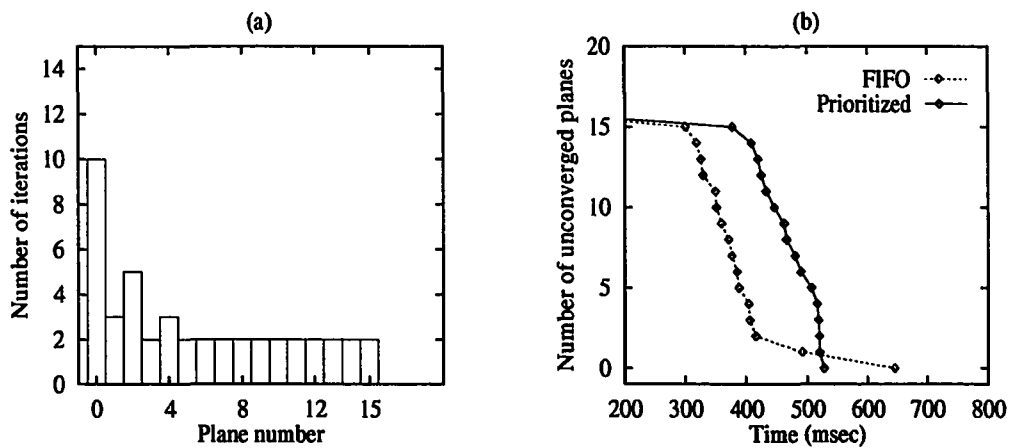


Figure 6.41: Multi-plane Stone (a) load index (b) concurrency index.

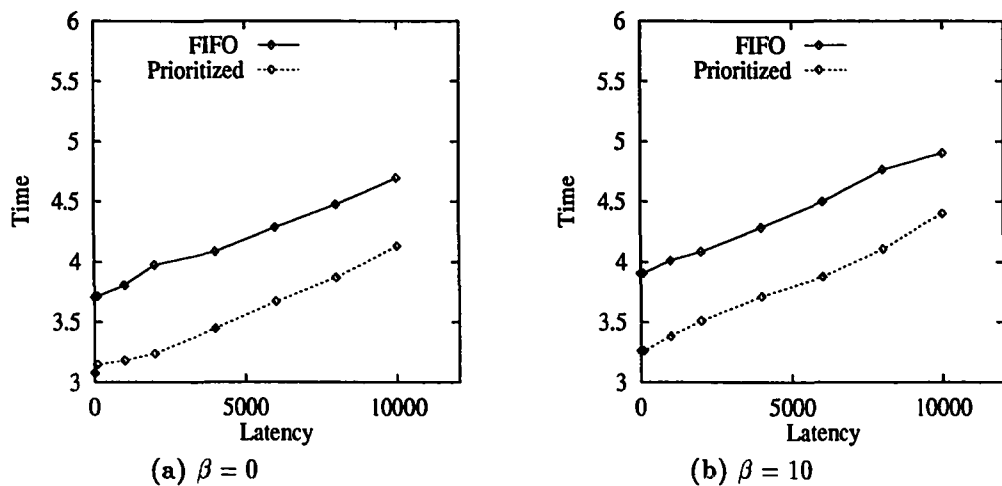


Figure 6.42: Multi-plane Stone : FIFO versus priority.

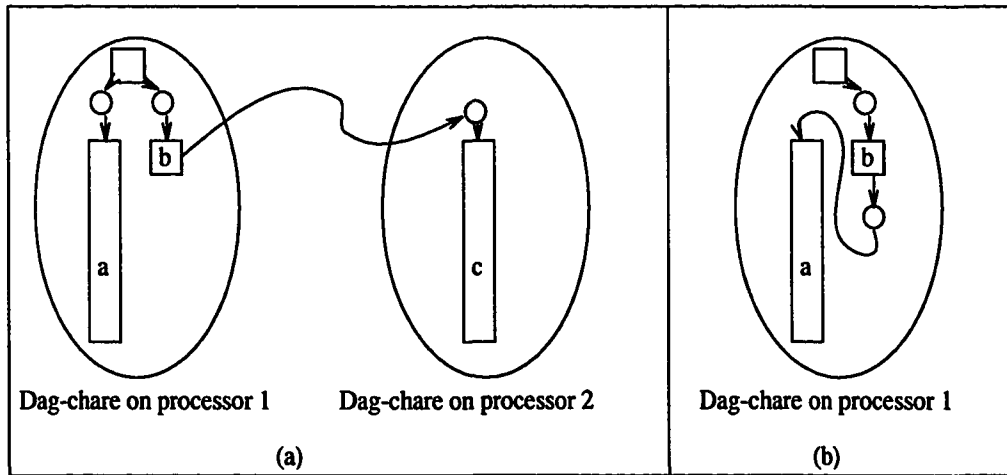


Figure 6.43: Greedy scheduling example (a) FIFO (b) forced static scheduling.

6.6.2.3 Adaptive scheduling is not always good

We have seen so far that adaptive scheduling achieves better performance than SPMD's static scheduling. However, there are some cases where the adaptive scheduling may perform poorer than the static one. Consider the example in Figure 6.43-a). Processor 1 has two computations a and b . These computations can be scheduled any order depending on a 's or b 's message (m_a and m_b respectively) arrives first. Processor 2 has a computation c which can start only after b has been completed. Assume that first m_a and shortly after m_b arrives. At the time m_a has arrived, processor 1 enables the computation a . The completion time for this case would be $t_a + t_b + t_c$ because a finishes, then b starts, after that c starts (t_i is the completion time of i). On the other hand, processor 1 can wait for m_b even though m_a arrives first. When m_b arrives, it executes b and sends the message to processor 2. Now a and c can be executed concurrently. The completion time for this case is $t_c + \max(t_a, t_b)$.

The message-driven programmer must be aware of such cases just as the SPMD programmer, who is aware of this critical path, and will force scheduling b before a . The message-driven programmer can also forego adaptive scheduling in such cases. Dagger does not preclude the expression of such scheduling decisions. For example, the computations a and b can be linearized by issuing the expect statement for m_a after completing b (Figure 6.43-b)).

6.7 Conclusion

We set out to determine and quantify the performance benefits of message-driven execution. Briefly, the conclusions can be summarized as follows. Overall, the message-driven execution imparts substantial performance benefits in a wide variety of cases. Message-driven execution derives these benefits from its ability to adaptively schedule messages. Its performance benefits do not come from the presence of communication latencies alone; they also arise because of the idle times in individual subcomputations due to the critical paths and load imbalances.

1. In fact, for applications where the sole benefit for message-driven execution is due to overlapping communication latencies, the advantage is limited to a bounded region because of the presence of overhead on one side and relative insignificance of the communication latencies on the other side. The extent of this region depends on the particular application and also depends on the technological parameters, such as the ratio of processor speed to communication latencies. As this ratio will probably become worse in the future, even this region is likely to increase its extent as faster processors are used.
2. Whenever there is irregularity and variance, either in the network latency or in computation times, etc., the message-driven execution adapts better and gives superior performance.
3. This benefit is substantial when one has one or more independent subcomputations with their own idle time.
4. The message-driven strategy also exploits the potential of a coprocessor better than an SPMD style, as it can effectively utilize the processor time freed up by the coprocessor.
5. Prioritized scheduling can be used to exercise control over the adaptive scheduling offered by message-driven execution, thus retaining the advantages of adapting message arrivals while ensuring progress on important parts of computation. Using interrupts for certain kind of messages such as reduction messages benefits the performance of message-driven programs for certain applications (Section 6.6.1). The coprocessors can be utilized to expedite the execution of such messages.

In the discussion below, some of the conclusions stated above will be substantiated and refined.

Multiple independent subcomputations

The most significant advantage of message-driven execution occurs when there are multiple independent subcomputations within a parallel program. Suppose two parallel algorithms, A1 and A2, are written in two modules, M1 and M2, such that the main program makes two independent calls to M1 and M2 simultaneously. There are three cases in which advantages of message-driven execution can be analyzed.

Case 1 is when the two modules are such that each individually has no idle time at zero communication latency (for example, the Mlib program with uniform subcomputations). If two such algorithms are composed using a message-driven model, then the performance advantage comes only if there is a significant communication latency. So the combined algorithm can tolerate the communication latency somewhat better than each individual algorithm. However, this is often tempered by the overhead introduced by the message-driven execution. In this case, there is a narrow range in which the message-driven execution can benefit.

Case 2 is when the two algorithms individually have variations in their subcomputations (i.e., the time for subcomputations on different processors that belong to the same algorithm may vary). Such situations may create unpredictable idle times on processors due to dependences in the algorithms (for example, the Mlib program with non-uniform subcomputations). In this case, message-driven execution helps, because it can adaptively switch between the algorithms depending.

Case 3 is when algorithm A1 as well as algorithm A2 (or at least one of them) has significant idle time (due to dependences) when run by itself even in presence of no communication latency (for example, the Wave program). In such cases, message-driven execution has a major advantage. It allows one to adaptively utilize the idle times within one algorithm for advancing the other algorithm and vice versa.

Essential performance benefits of message-driven execution

It is important to note the difference between the essential advantages of message-driven execution and stylistic or induced advantages of it. For example, in the Stone's method, the message-driven algorithm can be expressed by using extended SPMD by probing for the backward messages. This will give an identical effect of the message-driven algorithm. However, one

can not do this for concurrent reductions or for multiple library computations of the synthetic benchmark. In SPMD style, once another module is called, the caller is blocked until the callee returns. The only way to do this, in SPMD style, is to flatten the code (*i.e.*, combine code for multiple modules). Thus message-driven execution has an essential advantage that can not be duplicated in SPMD program when overlap across multiple modules is involved.

Benefits of coprocessor

Many of the current machines such as the Intel/Paragon provide a communication coprocessor. As the experiments in Section 6.3 show, the SPMD style cannot exploit the coprocessors as effectively as the message-driven execution.

Chapter 7

Conclusion

The objective of this thesis was to study message-driven execution as a mechanism for improving the efficiency of parallel programs. In Chapter 2, by analyzing various contexts, the potential benefits of message-driven execution was established, and it was observed that message-driven execution can potentially lead to improved performance either in presence of idle times of communication latencies or when multiple algorithms are composed where each algorithm has its own idle time due its critical path and load imbalances.

To express message-driven programs, a good language is needed. It was argued that why emulating message-driven style in SPMD programs is not a good idea. The Charm language was presented as an example of a message-driven language that can solve this problem. However, using Charm was not adequate for this study for two reasons: a) in pure message-driven execution style, it is difficult to express dependences and it is difficult to handle nondeterministic message arrivals, b) trace-driven simulation (which was a major aspect of the study) is not feasible (see Chapter 3). Therefore, a new notation, Dagger, was developed to express dependences between subcomputations and messages within a single object. The Dagger language has two basic constructs, namely *expect* and *when-blocks*, to express and enforce these dependences. In Charm, when a message is received, it triggers a task (the code associated with the entry-point). However, at that particular time, the task might not be ready to process the message due to dependences in the computation. The *expect* construct solves this problem. A subcomputation is triggered when a message for it arrives, and the message is allowed to trigger it.

Message-driven libraries are easier to compose into larger programs, and they do not require one to sacrifice performance in order to break a program into multiple modules. One can overlap the idle times across multiple independent module invocations.

The features of Dagger are also useful for trace-driven simulation programs. Simulation of message-driven programs is difficult because the order of instructions executed depends on many factors, and it changes as the machine characteristics change. If every instruction is to be actually simulated (execution-driven simulation) then the simulation becomes too expensive. Trace-driven simulation is a more efficient approach. However, trace-driven simulation is not possible from execution traces only, because the traces do not have sufficient information about the program to conduct simulation when the architectural conditions are different from the conditions under which the traces were gathered. Dagger solves this problem and allows us to conduct trace-driven simulations efficiently for a class of computations (see Chapter 5).

A simulation study was conducted involving a few synthetic benchmarks and a few taken from real applications. The performance benefits of message-driven execution are enhanced when there are variations in communication latencies or individual computations. The performance advantages of message-driven execution are not limited to communication latencies alone. If there are idle times in the individual algorithms, then message-driven programs are able to overlap and utilize processors better in such cases. It was also shown that a communication coprocessor can be exploited much better by a message-driven program than an SPMD program.

There are situations where message-driven programs may perform worse than SPMD programs. They typically involved greedy, adaptive scheduling of message-driven execution interfering with the critical path. Many such situations can be dealt with by use of priorities (prioritized scheduling strategies with a message-driven system). An extreme form of prioritization in the form of interrupts can be used in case of short computations on the critical path, such as reductions. Finally, it is certainly feasible to express a static SPMD like dependence in a message-driven language.

Criteria for designing message-driven algorithms were identified and were shown that they tend to be somewhat different from those for SPMD algorithms. In particular, in SPMD programs, the critical path is the sole important criterion, whereas for a message-driven algorithm

load imbalance also acquires substantial importance. This is because load imbalance affects the degree of overlap one can attain with another independent algorithm.

7.1 Future work

Future work that this research suggests is listed below:

Reduction of overhead As observed in the performance studies, the overhead per message in a message-driven system can be important in that it can offset the performance benefits of message-driven programs for cases where the average grain size of the computation (useful work per message) and the overhead are similar.

Both Dagger and Charm contribute to this overhead. At the Dagger level, the overhead comes from two sources: a) the overhead of searching for a message or a when-block instance in its queues for a given reference number and b) the overhead of the additional code produced by Dagger. At the Charm level, the basic sources of overhead per message are a) the allocation and deallocation of messages b) extra copy needed to transfer the data from the message to the user data structure (in SPMD style, especially if one is dealing with arrays, etc., the message can be received directly into user data structure if the receive call is issued before the message arrives). All these sources can be effectively dealt with. The current Dagger translator performs no optimization of the code it produces. The searches involved in the queues can be speeded up by employing hashing. The Charm message overhead can be overcome by exploiting the machine architecture and operating system to eliminate extra copying. Optimization of both Dagger and Charm can further enhance the benefits of message-driven execution.

Better language notation Dagger notation expresses the flow of control better than Charm and handles synchronization among messages and when-blocks by maintaining counts, flags etc (transparent to user). However, it is still a flat collection of when-blocks. The flow of computation is still not as clear as in SPMD programs. For a common special case where dependences form a structured partial order, one can think of a more intuitive notation. Indeed, such a notation, structured Dagger, is being developed to address these points.

We expect that the inherent performance benefits and compositionality advantages of message-driven execution will lead to its becoming a predominant style for writing parallel programs. The research in this thesis strongly suggests that programmers contemplating writing large parallel applications should seriously consider message-driven execution and message-driven languages.

Bibliography

- [1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.
- [2] A. Aho, J. Hopcraft, and J. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
- [3] R. Alverson, D. Callahan, and D. Cummnings et al. The TERA computer system. In *International Conference on Supercomputing*, pages 1–6, 1990.
- [4] D.A. Anderson, J.C. Tannehill, and R.H. Pletcher. *Computational Fluid Mechanics and Heat Transfer*. McGraw-Hill, 1984.
- [5] W. C. Athas and C. L. Seitz. Multicomputers: Message-passing concurrent computers. In *Computer*, volume Volume 21, No. 8, August 1988.
- [6] A.Yonezawa. *ABCL: An Object Oriented Concurrent System*. MIT Press, 1990.
- [7] S.B. Baden. Programming abstractions for dynamically partitioning localized scientific calculations. *SIAM J. Sci. Stat. Computation*, 12(1):145–157, January 1991.
- [8] A. Beguelin, J.J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. PVM and HeNCE: Tools for heterogeneous network computing. In J.J. Dongarra B.Tourancheau, editor, *Environments and Tools for Parallel Scientific Computing*, pages 139–153. Elsevier Science Publishers B. V., 1993.
- [9] B. Boothe and A. Ranade. Improved multithreading techniques for hiding latency in multi-processors. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 214–222, May 1992.

- [10] M.E. Braaten. Application of parallel computing in computational fluid dynamics: A review. Technical Report 89CRD121, Corporate Research and Development, GE, July 1989.
- [11] J. Bruno. Report on the feasibility of hypercube concurrent systems in computational fluid dynamics. Technical Report TR-86.7, RIACS, March 1986.
- [12] F.W. Burton and M.M. Huntbach. Virtual tree machines. *IEEE Trans. Computers*, c-33(3):278–280, March 1984.
- [13] R.H. Campell and A.N. Haberman. The specification of process synchronization by path expressions. In G. Goos and J. Hartmanis, editors, *Lecture Notes in Computer Science*, pages 89–102. Springer-Verlag, 1974.
- [14] N. Carriero and D.Gelernter. Linda in context. *Comm. ACM*, 32(4):444–458, 1989.
- [15] K.M. Chandy and C. Kesselman. C++: Compositionall parallel programming. Technical Report Caltech-CS-Tr-92-13, Department of Computer Sciencem California Institute of Technology, 1992.
- [16] The Charm 4.0 programming language manual. Department of Computer Science, University of Illinois at Urbana-Champaign, 1994.
- [17] A. Chien. *Concurrent Aggregates*. MIT Press, 1993.
- [18] A.T. Chronopoulos and C.W. Gear. On the efficient implementation of preconditioned s-step conjugate gradient methos on multiprocessors with memory hierarchy. *Parallel Computing*, (11):37–53, 1989.
- [19] J.S. Conery. *The AND/OR Process Model for Parallel Interpretation of Logic Programs*. PhD thesis, University of California, Irvine, June 1983.
- [20] D. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel programming in Split-C. Technical report, University of California, Berkeley.

- [21] D.E. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, May 1993.
- [22] W. Dally and et al. The J-Machine: A fine-grain concurrent computer. In *IFIP Congress*, 1989.
- [23] H. Davis, S. Goldschmidt, and J. Hennesy. Multiprocessor simulation and tracing using tango. In *Proceedings of the 1991 International Conference on Parallel Processing*, volume II, pages 99–107, St. Charles, IL, August 1991.
- [24] N.H. Decker, V.K. Naik, and M. Nicoules. Parallelization of implicit finite difference schemes in computational fluid dynamics. Technical Report 90-53, ICASE, NASA Langley Research Center, August 1990.
- [25] J.W. Demmel, M.T. Heath, and H. Van der Vorst. Parallel linear algebra. *Acta Numerica*, 2, 1993.
- [26] E.W. Dijkstra. Co-operating sequential processes. In F. Genuys, editor, *Notes on Structured Programming*, pages 43–112. Academic Press, New York, 1968.
- [27] E.W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, August 1975.
- [28] J. Dongarra, I. Duff Du Croz, and S. Hammarling. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 16(1):1–17, 1990.
- [29] M. Dubois, Faye A. Briggs, I. Patil, and M. Balakrishnan. Trace-driven simulations of parallel and distributed algorithms in multiprocessors. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 909–915, August 1986.
- [30] D. Culler et al. Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 164–175, 1991.

- [31] T.von Eicken et al. Active Messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 256–266, May 1992.
- [32] W. Fenton et al. Supporting machine independent programming on diverse parallel architectures. In *Proceedings of the 1991 International Conference on Parallel Processing*, pages 193–201, St. Charles, IL, August 1991.
- [33] P. Evripidou and J.L. Gaudiot. A decoupled data-driven architecture with vectors and macro actors. In G. Goos and J. Hartmanis, editors, *Proc. Joint Int. Conf. on Vector and Parallel Processing*, pages 39–50, Zurich, Switzerland, September 1990. Springer-Verlag.
- [34] R. Finkel and U.Manber. DIB - a distributed implementation of backtracking. *ACM TOPLAS*, 9:235–256, 1987.
- [35] HIGH PERFORMANCE FORTRAN FORUM. High performance fortran language specification, version 1.0 draft. Technical report, January 1993.
- [36] Message Passing Interface Forum. Document for a standard message passing interface. Technical Report CS-93-214, November 1993.
- [37] S. Frolund and G. Agha. Activation of concurrent objects by message sets. 1993.
- [38] E. Gaber. Vmpp: A practical tool for the development of portable and efficient programs for multiprocessors. *IEEE Trans. Parallel Distributed Systems*, 1(3):304–316, July 1990.
- [39] G. Golub and C. van Loan. *Matrix Computations*. John Hopkins University Press, Baltimore, 1983.
- [40] C. Gong, R. Gupta, and R. Melhem. Compilation techniques for optimizing communication on distributed-memory systems. In *Proceedings of the 1993 International Conference on Parallel Processing*, volume 2, pages 39–46, August 1993.
- [41] A.S. Grimshaw. *Mental : An Object Oriented Macro Data Flow System*. PhD thesis, University of Illinois at Urbana-Champaign, June 1988.
- [42] W.D. Gropp. Domain decomposition methods in computational fluid dynamics. Technical Report 91-20, ICASE, NASA Langley Research Center, February 1991.

- [43] M. Gupta. *Automatic Data Partitioning on Distributed Memory Multicomputers*. PhD thesis, Dept. of Computer Science, University of Illinois, 1992.
- [44] A. Gursoy and L.V. Kale. High-level support for divide-and-conquer parallelism. In *Proceedings of Supercomputing '91, Albuquerque, New Mexico*, pages 283–292, November 1991.
- [45] A. Gursoy and L.V. Kale. Simulating message driven programs. Technical Report 93-9, Parallel Programming Laboratory, Department of Computer Science, University of Illinois, Jul 1993.
- [46] A. Gursoy and L.V. Kale. Tolerating latency with Dagger. In *Proceedings of the Eight International Symposium on Computer and Information Sciences, Istanbul, Turkey*, pages 355–362, November 1993.
- [47] A. Gursoy and L.V. Kale. Dagger: Combining the benefits of synchronous and asynchronous communication styles. In *International Parallel Processing Symposium, Cancun, Mexico*, April 1994.
- [48] A. Gursoy, L.V. Kale, and S.P. Vanka. Unsteady fluid flow calculations using a machine independent parallel programming environment. In R. B. Pelz, A. Ecer, and J. Häuser, editors, *Parallel Computational Fluid Dynamics '92*, pages 175–185. North-Holland, 1993.
- [49] Robert H. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(2):501–538, October 1985.
- [50] P.B. Hansen. Distributed processes: A concurrent programming concept. *Communications of the ACM*, 21(11):934–941, November 1978.
- [51] F.H. Harlow and J.E. Welch. Numerical calculation of time dependent viscous incompressible flow of fluid with free surface. *Phys. of Fluids*, 8(112):2182–2189, 1965.
- [52] M.T. Heath and C.H. Romine. Parallel solution of triangular systems on distributed memory multiprocessors. *SIAM J. Sci. Stat. Computation*, 9(3):558–587, May 1988.
- [53] C.E. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8(3):323–364, June 1977.

- [54] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.
- [55] M.A. Holliday and C.S. Ellis. Accuracy of memory reference traces of computations in trace-driven simulation. *IEEE Trans. Parallel Distributed Systems*, 3(1):97–109, January 1992.
- [56] C. Houck and G. Agha. Hal: A high level actor languages and its distributed implementation. Technical Report UIUCDCS-R-92, Department of Computer Science, University of Illinois at Urbana-Champaign, September 1992.
- [57] J.M. Hsu and P. Banarjee. A message passing coprocessor for distributed memory multi-computers. In *Proceedings of Supercomputing 90*, pages 720–729, 1990.
- [58] J.M. Hsu and P. Banarjee. Performance measurement and trace driven simulation of parallel cad and numeric applications on a hypercube multicomputer. *IEEE Trans. Parallel Distributed Systems*, 3(4):398–412, July 1992.
- [59] K.L. Johnson. The impact of communication locality on large scale multiprocessor performance. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 392–402, May 1992.
- [60] L.V. Kale. The Chare kernel parallel programming language and system. In *Proceedings of the 1990 International Conference on Parallel Processing*, volume II, pages 17–25, St. Charles, IL, August 1990.
- [61] L.V. Kale. A tutorial introduction to Charm. Technical Report 92-6, Parallel Programming Laboratory, Department of Computer Science, University of Illinois, 1992.
- [62] L.V. Kale and A. Gursoy. Performance benefits of message driven execution. In *Proceedings of the Intel SUG*, pages –, October 1993.
- [63] L.V. Kale and B. Ramkumar. *The Reduce-OR-Process Model for Parallel Logic Programming on Nonshared Memory Machines*. John Wiley, June 1992. Editors: M. Wise and P. Kacsuk.

- [64] J. Kim and P. Moin. Application of a fractional step method to incompressible navier-stokes equations. *J. Comp. Phys*, 59:308–323, 1985.
- [65] A.C. Klaiber and J.L. Frankel. Comparing data-parallel and message-passing paradigms. In *Proceedings of the 1993 International Conference on Parallel Processing*, volume 2, pages 11–20, August 1993.
- [66] E. Kornkven and L.V. Kale. Dynamic adaptive scheduling in an implementation of a data parallel language. Technical Report 92-10, Parallel Programming Laboratory, Department of Computer Science, University of Illinois, October 1992.
- [67] V. Kumar and V.N. Rao. Parallel depth first search, part 2: Analysis. *International Journal of Parallel Programming*, pages 501–519, December 1987.
- [68] A.W. Kwan, L. Bic, and D.D. Gajski. Improving parallel program performance using critical path analysis. In A. Nicolau D. Gelernter and D. Padua, editors, *Languages and Compilers for Parallel Computing*. MIT Press, Massachusetts, 1990.
- [69] P. Lee and Tzung-Bow. Compiling efficient programs for tightly-coupled distributed memory computers. In *Proceedings of the 1993 International Conference on Parallel Processing*, volume 2, pages 39–46, August 1993.
- [70] O.C. Maquelin. ADAM: a course-grain dataflow architecture that addresses the load balancing and throttling problems. In G. Goos and J. Hartmanis, editors, *Proc. Joint Int. Conf. on Vector and Parallel Processing*, pages 265–276, Zurich, Switzerland, September 1990. Springer-Verlag.
- [71] S. Murer and P. Farber. Code generation for multi-threaded architectures from dataflow graphs. In J.L. Gaudiot M. Cosnard, K. Ebcioglu, editor, *Architectures and Compilation Techniques for Fine Grain Parallelism*, pages 77–90. North-Holland, 1993.
- [72] W. Najjar and J.L. Gaudiot. A hierarchical data-driven model for multi grid problem solving. In E. Gelenbe, editor, *Proc. Int. Sym on High Performance Computer Systems*, pages 67–75, Paris-France, December 1987. Elsevier (North-Holand).

- [73] P.A. Nelson and L. Synder. Programming paradigms for nonshared memory parallel computers. In D.B.Gannon L.H.Jamieson and R.J. Douglas, editors, *The Characteristics of Parallel Algorithms*. MIT Press, 1987.
- [74] R.S. Nikhil. A multithreaded implementation of Id using P-RISC graphs. In *Sixth Annual Workshop on Languages and Compilers for Parallel Computing*, Portland, Oregon, August 1993.
- [75] J.M. Ortega. *Introduction to Parallel and Vector Solution of Linear Systems*. Plenum, 1988.
- [76] G.R. Rao and Z. Parasvekas. Compiling for dataflow software pipelining. In Alexandru Nicolau David Gelernter and David Padua, editors, *Languages and Compilers for Parallel Computing*. MIT Press, Massachusetts, 1990.
- [77] D.A. Reed and R.M. Fujimoto. *Multicomputer Networks: Message-Based Parallel Processing*. MIT Press, 1987.
- [78] D.A. Reed and M.L. Patrick. Parallel iterative solution of sparse linear systems: Models and architectures. *Parallel Computing*, (2):45–67, 1985.
- [79] J.Van Rosendale. Minimizing inner product data dependencies in conjugate gradient iteration. In *Proceedings of the 1983 International Conference on Parallel Processing*, pages 44–46, August 1993.
- [80] M. Rosing and R. Schnabel. Flexible language constructs for large parallel programs. Technical Report 93-29, ICASE NASA Langley Research Center, 1993.
- [81] Y. Saad and M.H. Schultz. Data communications in parallel architectures. *Parallel Computing*, (11):131–150, 1989.
- [82] V. Saletore. *Machine Independent Parallel Execution of Speculative Computations*. PhD thesis, Dept. of Computer Science, University of Illinois, 1990.
- [83] J.H. Saltz, V.K. Naik, and D.M. Nicol. Reduction of the effects of the communication delays in scientific algorithms on message passing mimd architectures. *SIAM J. Sci. Stat. Computation*, 8(1):s118–s134, January 1987.

- [84] S.R. Sarukkai and A.D. Malony. Perturbation analysis of high level instrumentation for spmd programs. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 44–53, May 1993.
- [85] T. Shimuzu, T. Horie, and H. Ishihata. Low-latency message communication support for teh ap1000. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 288–297, May 1992.
- [86] A. Smith. Cache memories. *ACM Comput. Surveys*, 14(3):473–530, September 1992.
- [87] B. Smith. The architecture of the HEP. In Kowalik, editor, *Parallel MIMD Computation: HEP Supercomputer and Applications*. MIT Press, Cambridge, MA, 1985.
- [88] H.L. Stone. Iterative solution of implicit approximations of multidimensional partial differential equations. *SIAM J.Numer.Anal.*, 5(3):530–558, September 1968.
- [89] Q.F. Stout. Supporting divide-and-conquer algorithms for image processing. *Journal of Parallel and Distributed Computing*, 4:95–115, 1987.
- [90] R.A. Sweet, W.L. Briggs, S. Oliveira, J.L. Porsche, and T. Turnbull. Ffts and three-dimensional poisson solvers for hypercubes. *Parallel Computing*, (17):121–131, 1991.
- [91] C. Tomlinson and V. Singh. Inheritance and synchronization with enabled-sets. In *ACM OOPSLA*, pages 103–112, 1989.
- [92] B.K. Totty and D.A. Reed. Dynamic object management for distributed data structures. In *Proceedings of Supercomputing '92, Minneapolis*, pages 692–701, November 1992.
- [93] C.E. Wu, J-P. Prost, and C. Benveniste. The design of a timing simulator for distributed applications. In *Proceedings of 1992 International Conference on Parallel and Distributed Systems, Taiwan, R.O.C*, pages 50–57, December 1992.
- [94] H. Wu and L.E. Thorelli. Extending dataflow principles for multiprocessing. In *Proceedings of the 1990 International Conference on Parallel Processing*, volume 2, pages 53–60, August 1990.

Vita

Attila Gürsoy was born on November 21, 1964 in Malatya, Turkey. He received the Bachelor of Science degree in Computer Science from Middle East Technical University, Ankara, Turkey in June 1986. He then continued into graduate school at Bilkent University, Ankara, Turkey. In June 1988 he earned Master of Science degree in Computer Science. He continued his research in parallel programming environments under Prof. L.V. Kale and earned the Doctor of Philosophy degree in Computer Science from University of Illinois at Urbana-Champaign in April 1994.